Breadth First Search
Depth First Search

Week-09, Lecture-02

Course Code: CSE221

Course Title: Algorithms

Program: B.Sc. in CSE

Course Teacher: Tanzina Afroz Rimi

Designation: Lecturer

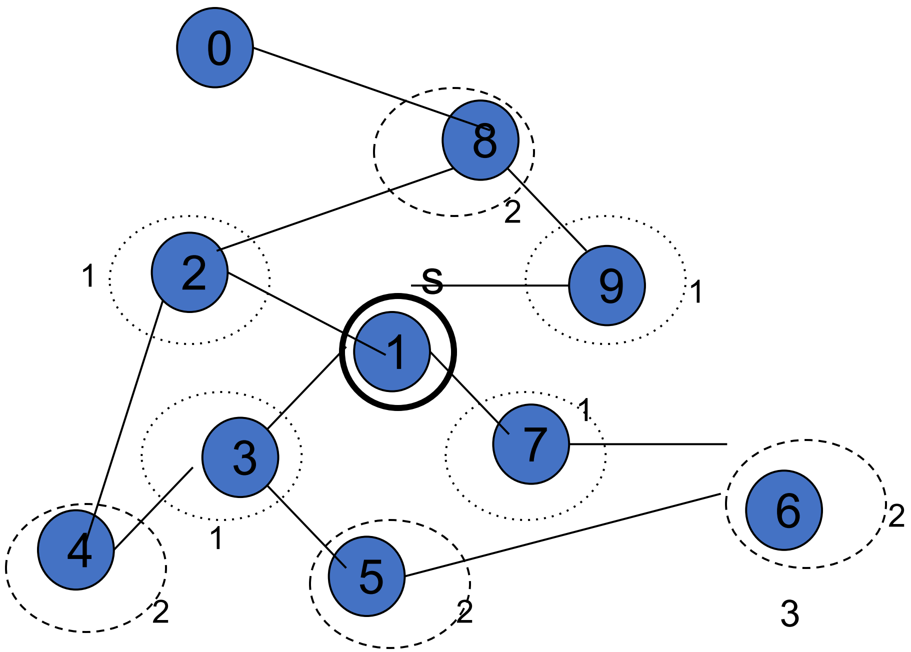
Email: tanzinaafroz.cse@diu.edu.bd

Graph Traversal

- Application example
  - Given a graph representation and a vertex **s** in the graph
  - Find paths from **s** to other vertices
- Two common graph traversal algorithms
  - Breadth-First Search (BFS)
    - Find the shortest paths in an unweighted graph
  - Depth-First Search (DFS)
    - Topological sort
    - Find strongly connected components

BFS and Shortest Path Problem

- Given any source vertex ***s***, BFS visits the other vertices at increasing distances away from s. In doing so, BFS discovers paths from s to other vertices
- What do we mean by "distance"? The number of edges on a path from s



Example

Consider s=vertex 1

Nodes at distance 1?
2, 3, 7, 9

Nodes at distance 2?
8, 6, 5, 4

Nodes at distance 3?
0

Graph Searching

- Given: a graph G = (V, E), directed or undirected
- Goal: methodically explore every vertex and every edge
- Ultimately: build a tree on the graph
  - Pick a vertex as the root
  - Choose certain edges to produce a tree
  - Note: might also build a *forest* if graph is not connected

Breadth-First Search

- "Explore" a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find ("discover") its children, then their children, etc.

Breadth-First Search

- Every vertex of a graph contains a color at every moment:
  - White vertices have not been discovered
    - All vertices start with white initially
  - Grey vertices are discovered but not fully explored
    - They may be adjacent to white vertices
  - Black vertices are discovered and fully explored
    - They are adjacent only to black and gray vertices
- Explore vertices by scanning adjacency list of grey vertices

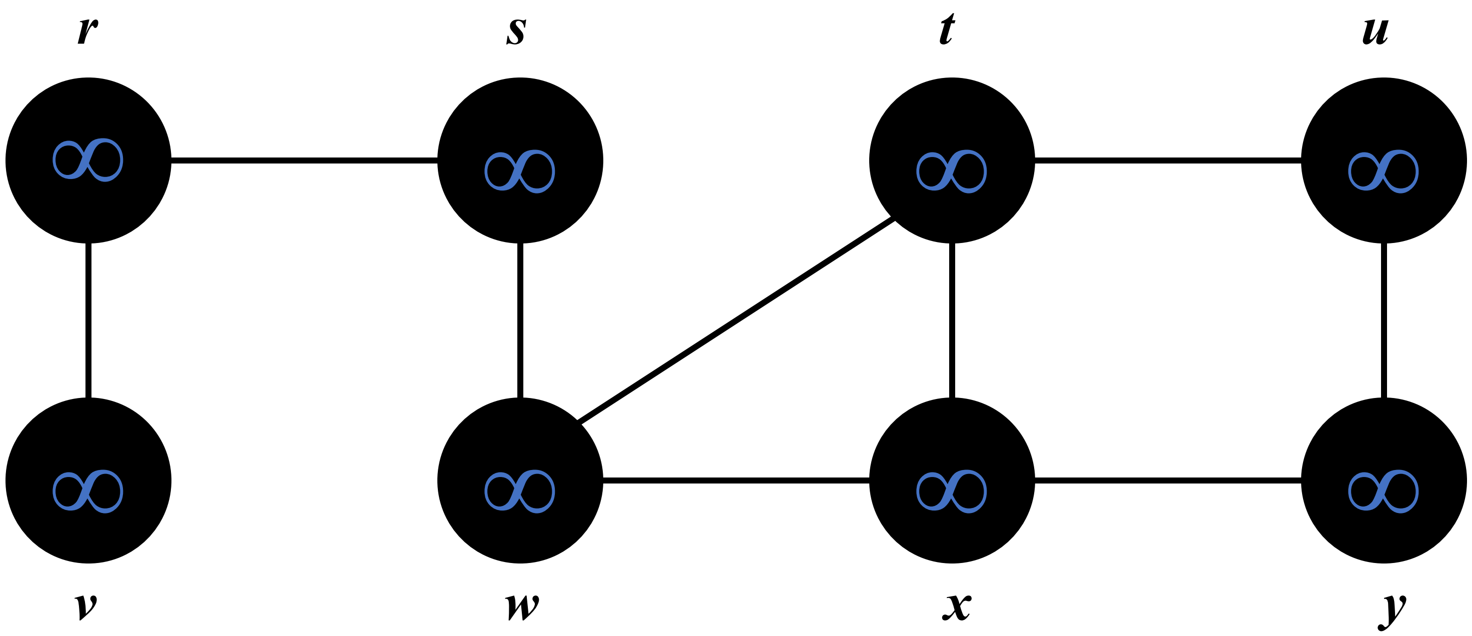
Breadth-First Search: The Code

```
Data: color[V], prev[V],d[V]

BFS(G) // starts from here
{
   for each vertex u ∈
  V-{s}
   {
      color[u]=WHITE;
   prev[u]=NIL;
   d[u]=inf;
   }
   color[s]=GRAY;
  d[s]=0; prev[s]=NIL;
  Q=empty;
  ENQUEUE(Q,s);
```

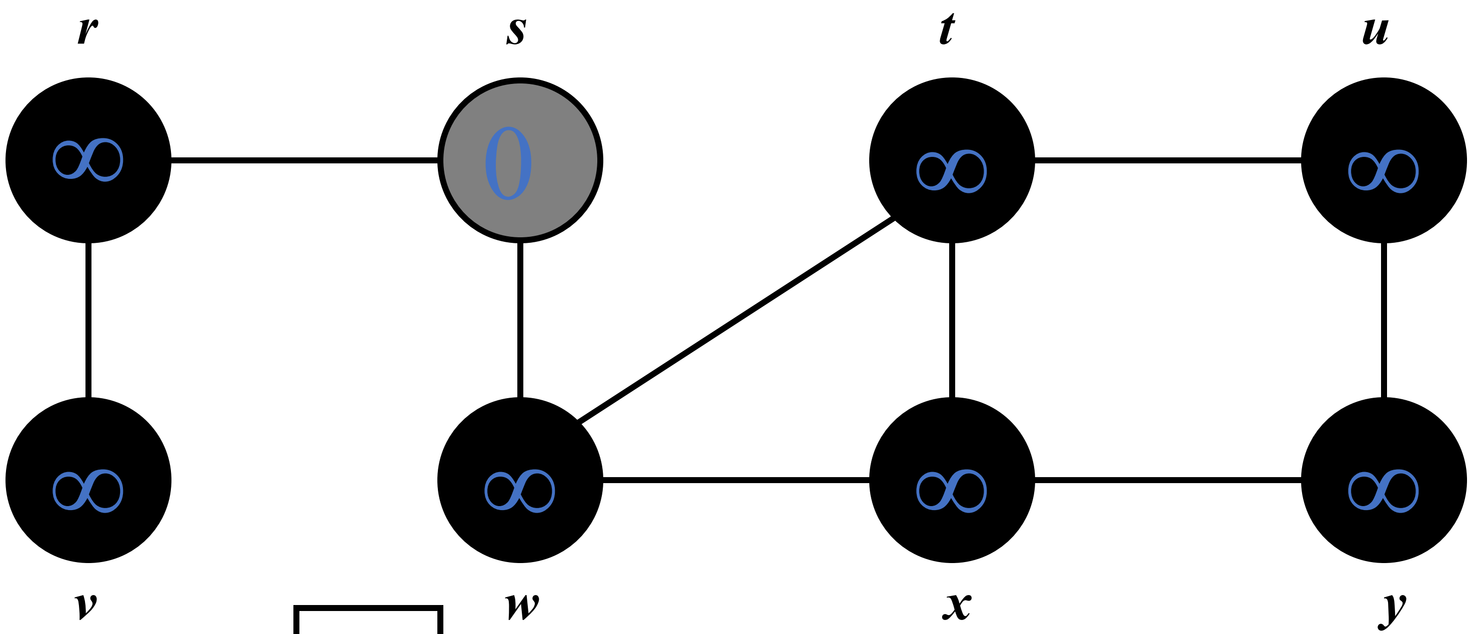
```
 While(Q not empty)
 {
   u = DEQUEUE(Q);
   for each v ∈ adj[u]{
     if (color[v] == WHITE){
         color[v] = GREY;
         d[v] = d[u] + 1;
         prev[v] = u;
         Enqueue(Q, v);
     }
   }
   color[u] = BLACK;
 }
}
```

Breadth-First Search: Example



Vertex	r	s	t	u	v	w	x	y
color	W	W	W	W	W	W	W	W
d	∞	∞	∞	∞	∞	∞	∞	∞
prev	nil	nil	nil	nil	nil	nil	nil	nil

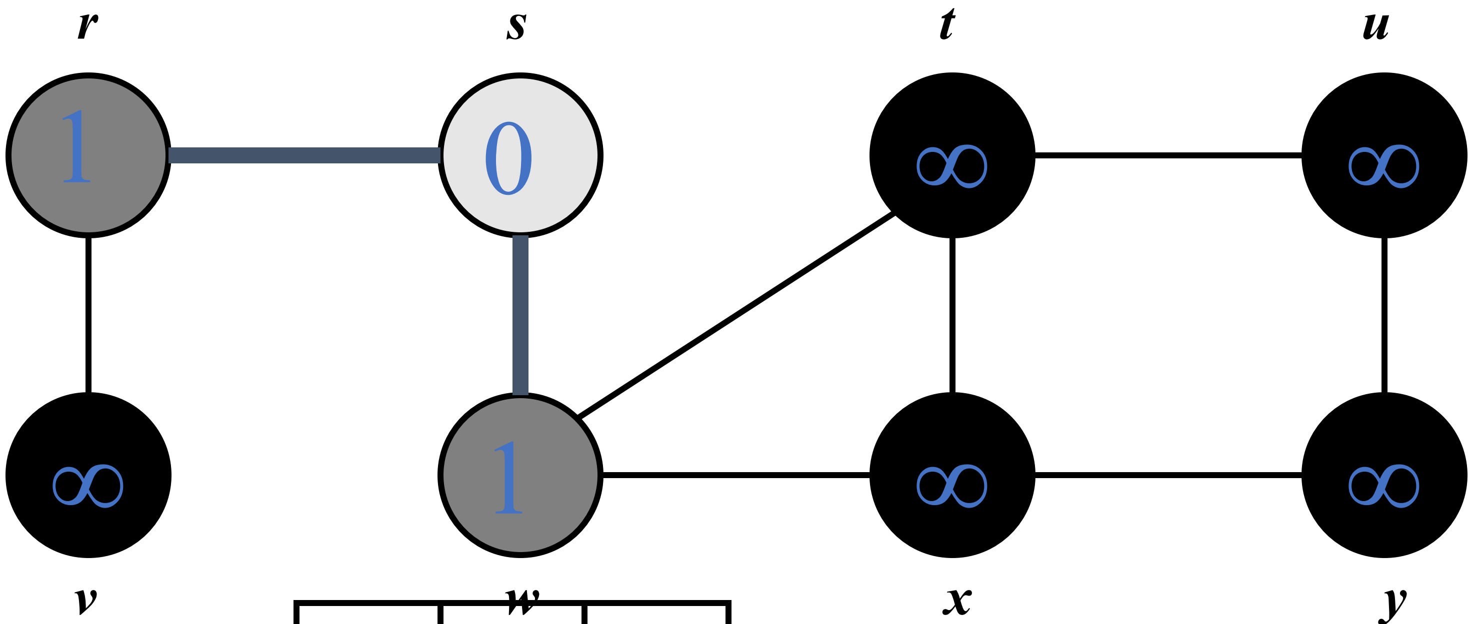
Breadth-First Search: Example



Q: $\boxed{s}$

vertex	r	s	t	u	v	w	x	y
Color	W	**G**	W	W	W	W	W	W
d	∞	**0**	∞	∞	∞	∞	∞	∞
prev	nil	**nil**	nil	nil	nil	nil	nil	nil

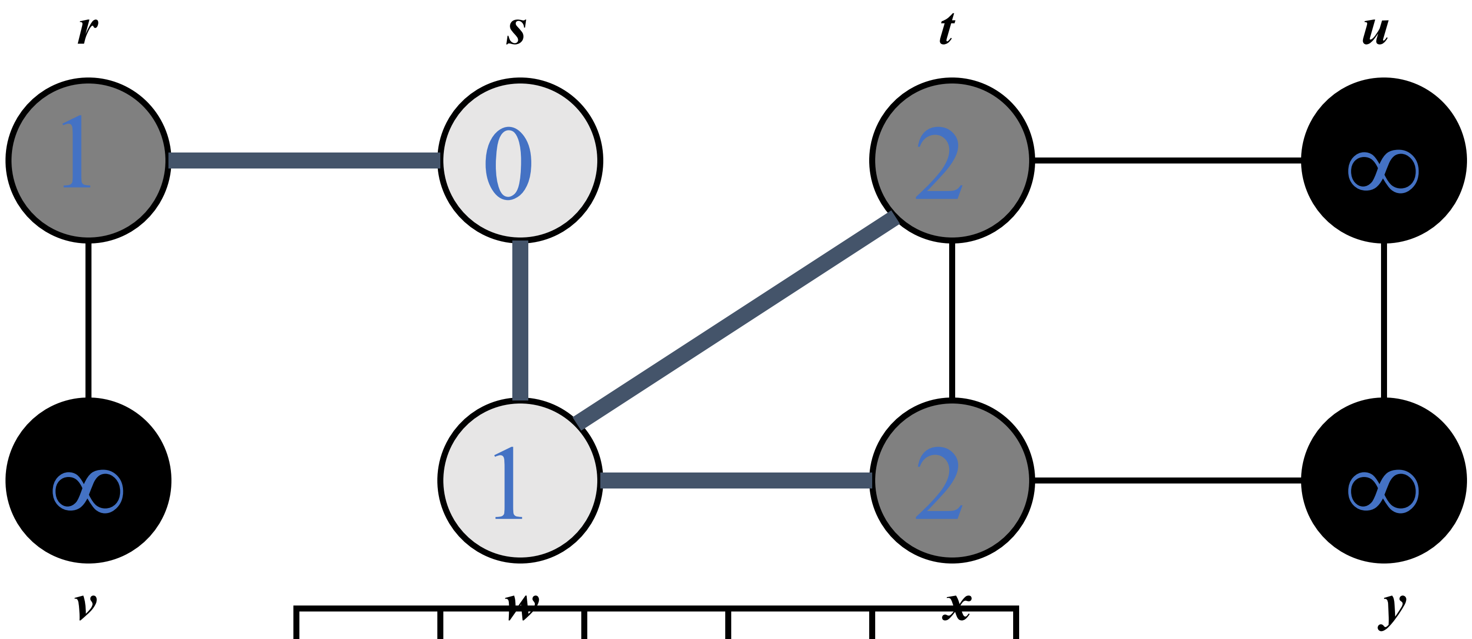
Breadth-First Search: Example



Q: | *s* | *w* | *r* |

vertex	r	s	t	u	v	w	x	y
Color	G	**B**	W	W	W	G	W	W
d	1	**0**	∞	∞	∞	1	∞	∞
prev	s	**nil**	nil	nil	nil	s	nil	nil

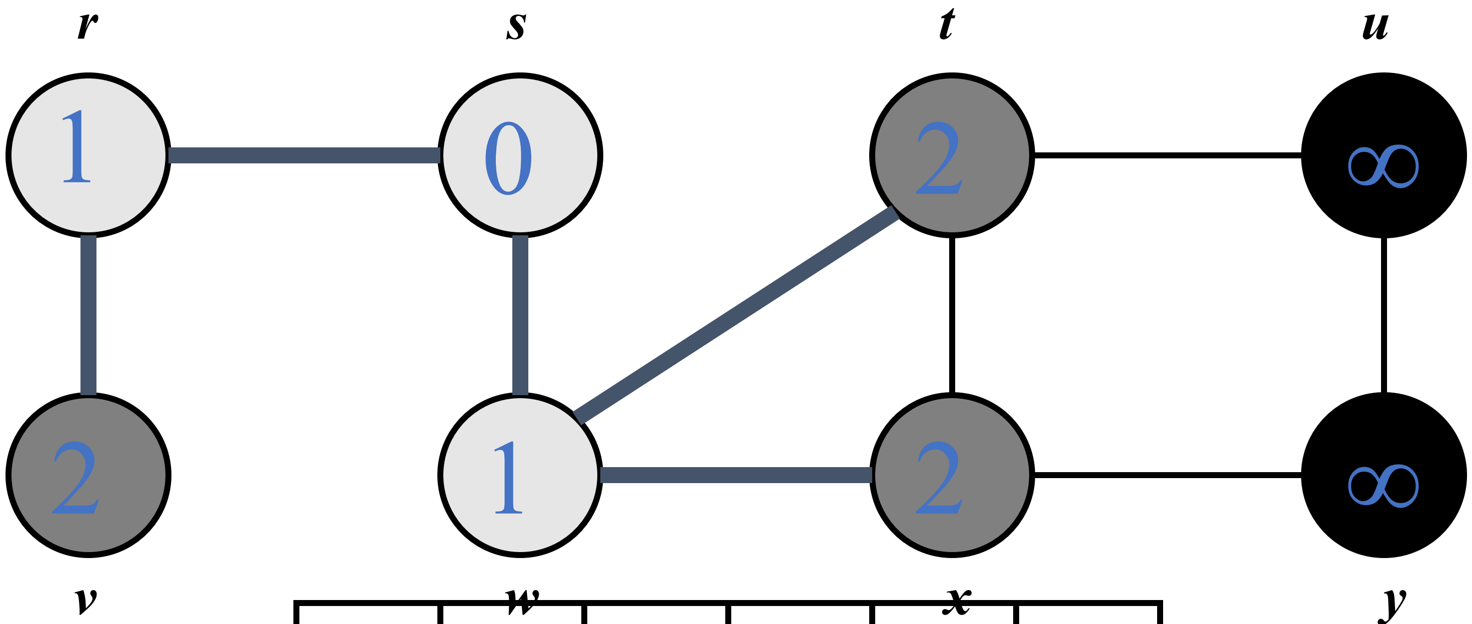
Breadth-First Search: Example



Q:	s	w	r	t	x

vertex	r	s	t	u	V	w	X	y
Color	G	B	**G**	W	W	**B**	**G**	W
d	1	0	**2**	∞	∞	**1**	**2**	∞
prev	s	nil	**w**	nil	nil	**s**	**w**	nil

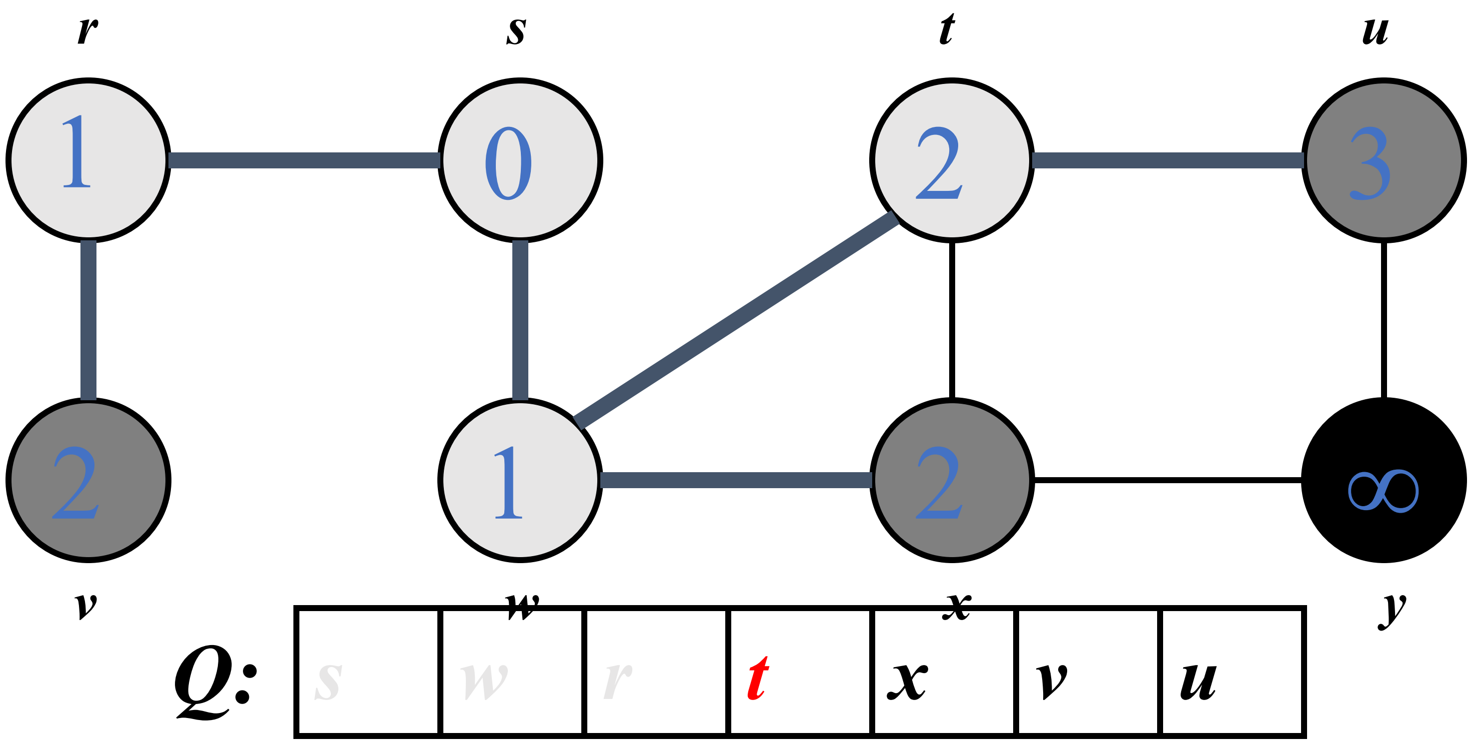
Breadth-First Search: Example



Q: | *s* | *w* | *r* | *t* | *x* | *v* |

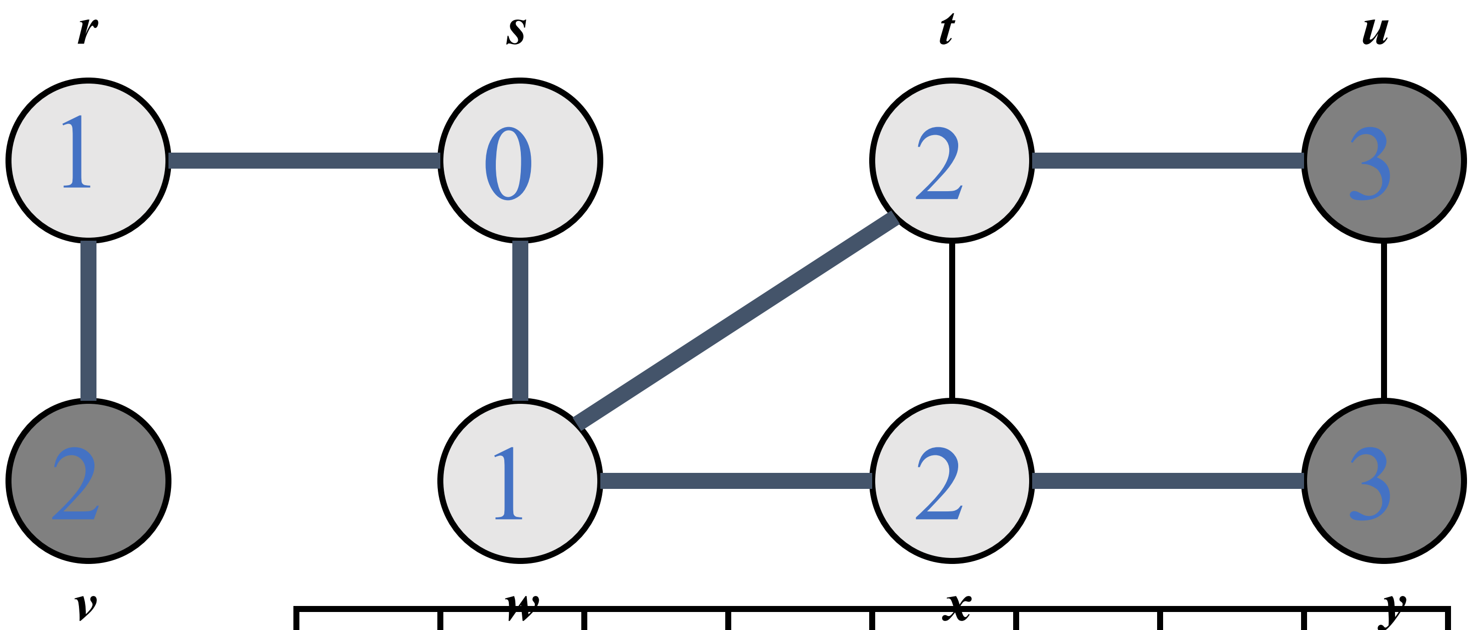
vertex	r	s	t	u	v	w	x	y
Color	B	B	G	W	G	B	G	W
d	1	0	2	∞	2	1	2	∞
prev	s	nil	w	nil	r	s	w	nil

Breadth-First Search: Example



vertex	r	s	t	u	v	w	x	y
Color	B	B	**B**	G	G	B	G	W
d	1	0	**2**	3	2	1	2	∞
prev	s	nil	**w**	t	r	s	w	nil

Breadth-First Search: Example

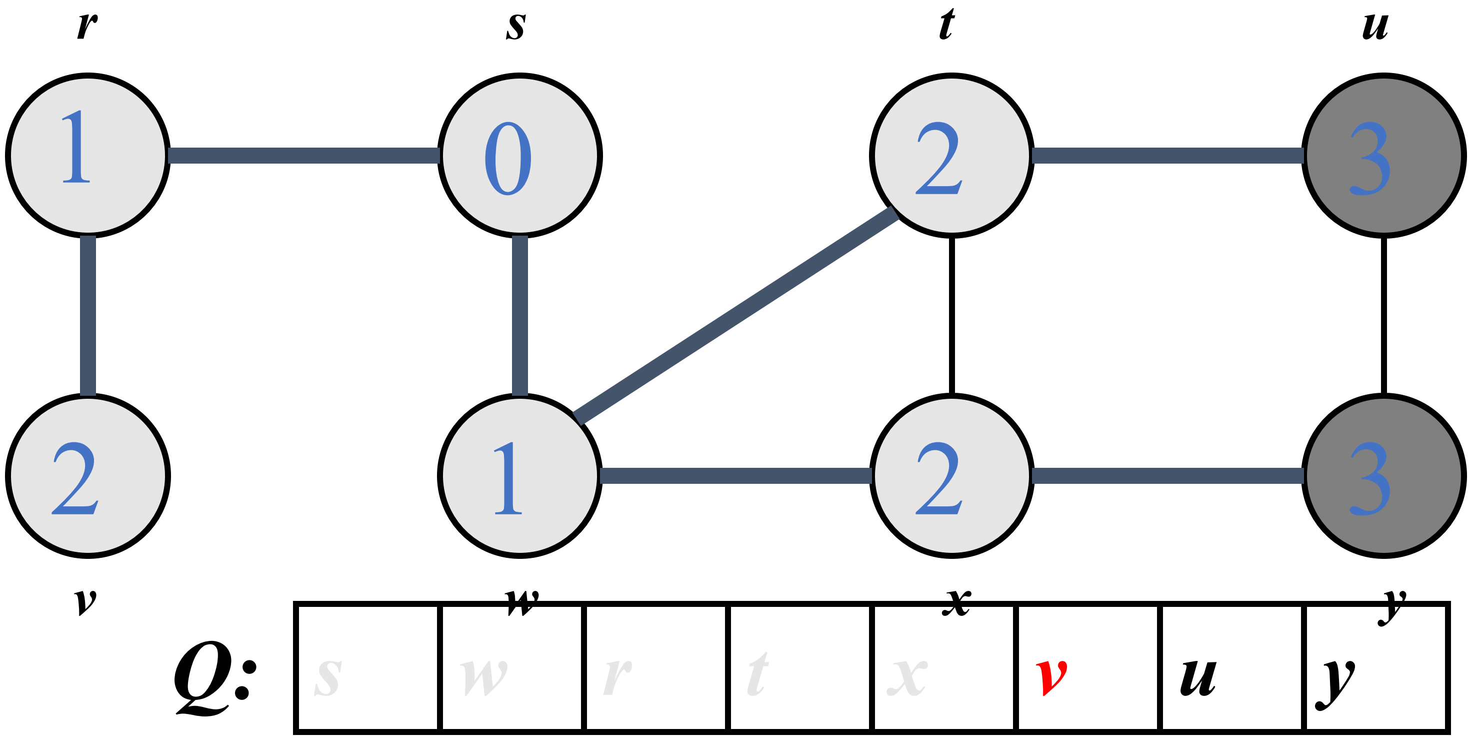


Q:

s	*w*	*r*	*t*	*x*	*v*	*u*	*y*

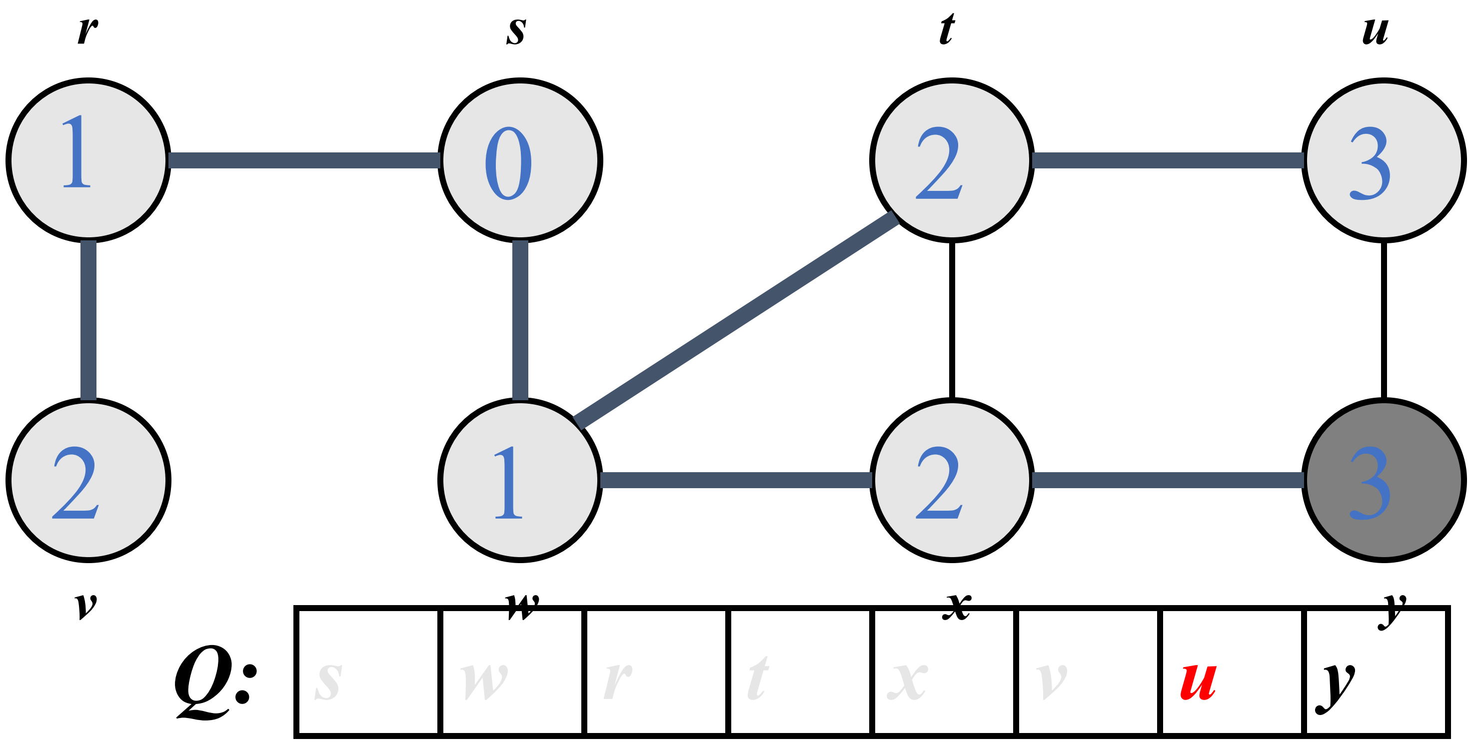
vertex	r	s	t	u	v	w	x	y
Color	B	B	B	G	G	B	B	G
d	1	0	2	3	2	1	2	3
prev	s	nil	w	t	r	s	w	x

Breadth-First Search: Example



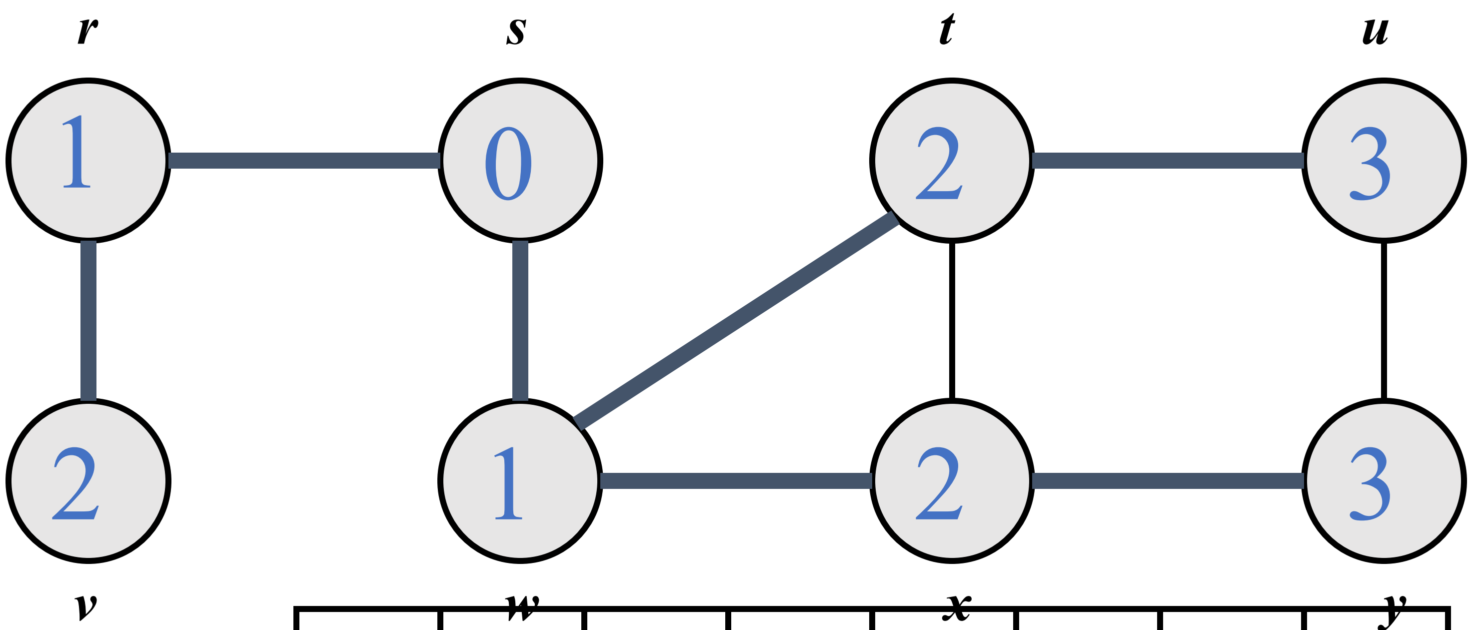
vertex	r	s	t	u	v	w	x	y
Color	B	B	B	G	**B**	B	B	G
d	1	0	2	3	**2**	1	2	3
prev	s	nil	w	t	**r**	s	w	x

Breadth-First Search: Example



vertex	r	s	t	u	v	w	x	y
Color	B	B	B	**B**	B	B	B	G
d	1	0	2	**3**	2	1	2	3
prev	s	nil	w	**t**	r	s	w	x

Breadth-First Search: Example



Q:

s	*w*	*r*	*t*	*x*	*v*	*u*	*y*

vertex	r	s	t	u	v	w	x	y
Color	B	B	B	G	B	B	B	**B**
d	1	0	2	3	2	1	2	**3**
prev	s	nil	w	t	r	s	w	**x**

BFS: The Code (again)

```
Data: color[V], prev[V],d[V]

BFS(G) // starts from here
{
   for each vertex u ∈
  V-{s}
   {
      color[u]=WHITE;
   prev[u]=NIL;
   d[u]=inf;
   }
   color[s]=GRAY;
  d[s]=0; prev[s]=NIL;
  Q=empty;
  ENQUEUE(Q,s);
```

```
 While(Q not empty)
 {
   u = DEQUEUE(Q);
   for each v ∈ adj[u]{
     if (color[v] == WHITE){
         color[v] = GREY;
         d[v] = d[u] + 1;
         prev[v] = u;
         Enqueue(Q, v);
     }
   }
   color[u] = BLACK;
 }
}
```

Breadth-First Search: Print Path

```
Data: color[V], prev[V],d[V]
```

```
Print-Path(G, s, v)
{
  if(v==s)
    print(s)
   else if(prev[v]==NIL)
    print(No path);
  else{
    Print-Path(G,s,prev[v]);
    print(v);
  }
}
```

Amortized Analysis

- Stack with 3 operations:
  - Push, Pop, Multi-pop
- What will be the complexity if “n” operations are performed?

BFS: Complexity

```
Data: color[V], prev[V],d[V]

BFS(G) // starts from here
{
   for each vertex u ∈
  V-{s}
   {
      color[u]=WHITE;
   prev[u]=NIL;
   d[u]=inf;
   }
   color[s]=GRAY;
  d[s]=0; prev[s]=NIL;
  Q=empty;
  ENQUEUE(Q,s);
```

O(V)

```
While(Q not empty)
{
  u = DEQUEUE(Q);
  for each v ∈ adj[u]{
   if(color[v] == WHITE){
        color[v] = GREY;
        d[v] = d[u] + 1;
        prev[v] = u;
        Enqueue(Q, v);
    }
  }
  color[u] = BLACK;
 }
}
```

u vertex, but only once
(Why?)

O(V)

What will be the running time?

Total running time: O(V+E)

Breadth-First Search: Properties

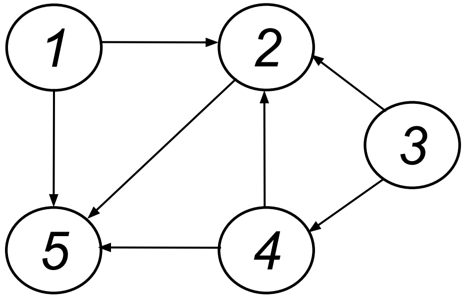
- BFS calculates the *shortest-path distance* to the source node
  - Shortest-path distance $\delta(s,v)$ = minimum number of edges from s to v, or ∞ if v not reachable from s
  - Proof given in the book (p. 472-5)
- BFS builds *breadth-first tree*, in which paths to root represent shortest paths in G
  - Thus can use BFS to calculate shortest path from one vertex to another in O(V+E) time

Application of BFS

- Find the shortest path in an undirected/directed unweighted graph.
- Find the bipartiteness of a graph.
- Find cycle in a graph.
- Find the connectedness of a graph.

Depth-First Search

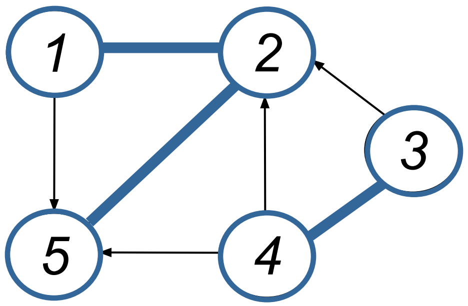
Depth-First Search



- **Input:**
  - G = (V, E) (No source vertex given!)
- **Goal:**
  - Explore the edges of G to "discover" every vertex in V starting at the most current visited node
  - Search may be repeated from multiple sources
- **Output:**
  - 2 **timestamps** on each vertex:
    - d[v] = discovery time
    - f[v] = finishing time (done with examining v's adjacency list)
  - Depth-first forest

Depth-First Search

- Search "deeper" in the graph whenever possible
- Edges are explored out of the most recently discovered vertex v that still has unexplored edges



- *After all edges of v have been explored, the search "backtracks" from the parent of v*
- *The process continues until all vertices reachable from the original source have been discovered*
- *If undiscovered vertices remain, choose one of them as a new source and repeat the search from that vertex*
- *DFS creates a "depth-first forest"*

DFS Additional Data Structures

- Global variable: time-stamp
  - Incremented when nodes are discovered or finished
- color[u] – similar to BFS
  - White before discovery, gray while processing and black when finished processing
- prev[u] – predecessor of u
- d[u], f[u] – discovery and finish times

Depth-First Search: The Code

```
Data: color[V], time,
   prev[V],d[V], f[V]
DFS(G) // where prog starts
{                                    Initialize
  for each vertex u ∈ V
  {
     color[u] = WHITE;
  prev[u]=NIL;
  f[u]=inf; d[u]=inf;
  }
  time = 0;
  for each vertex u ∈ V
    if (color[u] == WHITE)
        DFS_Visit(u);
}
```

```
DFS_Visit(u)
{
   color[u] = GREY;
   time = time+1;
   d[u] = time;
   for each v ∈ Adj[u]
   {
      if(color[v] == WHITE){
      prev[v]=u;
         DFS_Visit(v);}
  }
   color[u] = BLACK;
   time = time+1;
   f[u] = time;
}
```

Depth-First Search: The Code

```
Data: color[V], time,
  prev[V],d[V], f[V]
DFS(G) // where prog starts
{
   for each vertex u ∈ V
   {
      color[u] = WHITE;
   prev[u]=NIL;
   f[u]=inf; d[u]=inf;
   }
   time = 0;
   for each vertex u ∈ V
     if (color[u] == WHITE)
         DFS_Visit(u);
}
```

```
DFS_Visit(u)
{
   color[u] = GREY;
   time = time+1;
   d[u] = time;
   for each v ∈ Adj[u]
   {
      if(color[v] == WHITE){
      prev[v]=u;
         DFS_Visit(v);}
   }
   color[u] = BLACK;
   time = time+1;
   f[u] = time;
}
```


What does `u[d]` *represent?*

Depth-First Search: The Code

```
Data: color[V], time,
  prev[V],d[V], f[V]
DFS(G) // where prog starts
{
   for each vertex u ∈ V
   {
      color[u] = WHITE;
      prev[u]=NIL;
      f[u]=inf; d[u]=inf;
   }
   time = 0;
   for each vertex u ∈ V
     if (color[u] == WHITE)
         DFS_Visit(u);
}
```

```
DFS_Visit(u)
{
   color[u] = GREY;
   time = time+1;
   d[u] = time;
   for each v ∈ Adj[u]
   {
      if(color[v] == WHITE){
      prev[v]=u;
         DFS_Visit(v);}
   }
   color[u] = BLACK;
   time = time+1;
   f[u] = time;
}
```

What does `f[d]` *represent?*

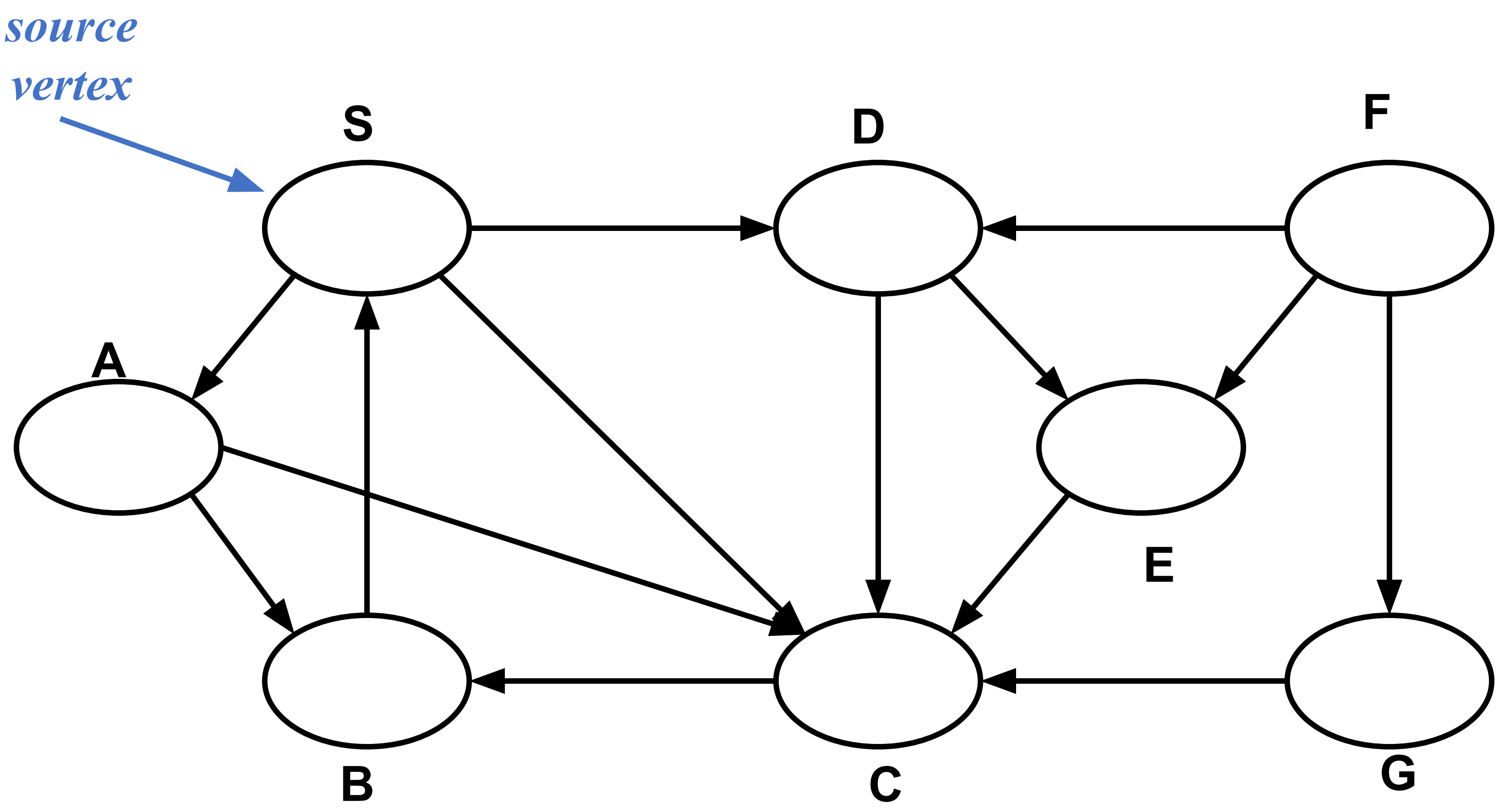
Depth-First Search: The Code

```
Data: color[V], time,
  prev[V],d[V], f[V]
DFS(G) // where prog starts
{
   for each vertex u ∈ V
   {
      color[u] = WHITE;
   prev[u]=NIL;
   f[u]=inf; d[u]=inf;
   }
   time = 0;
   for each vertex u ∈ V
     if (color[u] == WHITE)
         DFS_Visit(u);
}
```

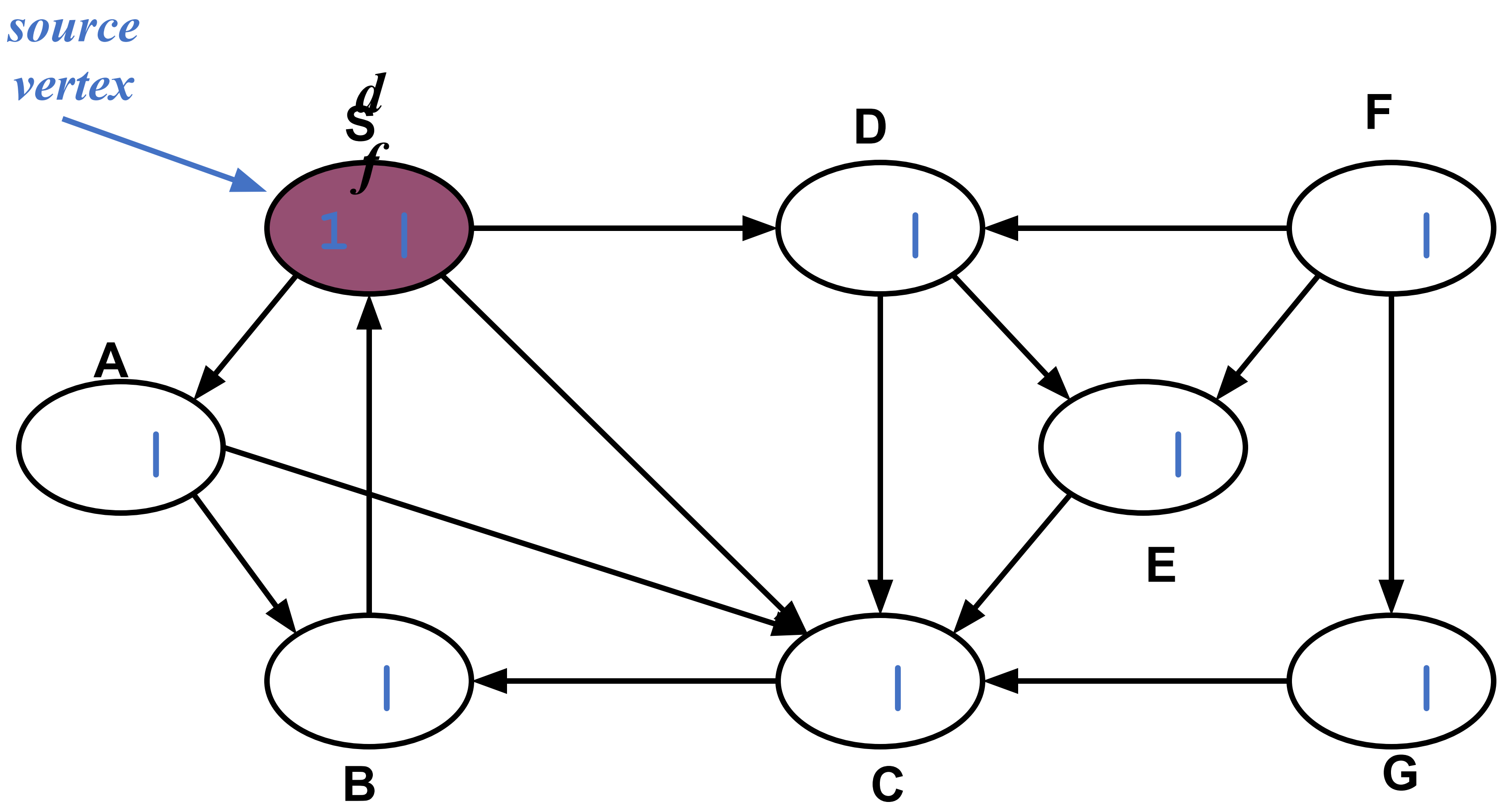
```
DFS_Visit(u)
{
   color[u] = GREY;
   time = time+1;
   d[u] = time;
   for each v ∈ Adj[u]
   {
      if(color[v] == WHITE){
      prev[v]=u;
         DFS_Visit(v);
   }}
   color[u] = BLACK;
   time = time+1;
   f[u] = time;
}
```

Will all vertices eventually be colored black?

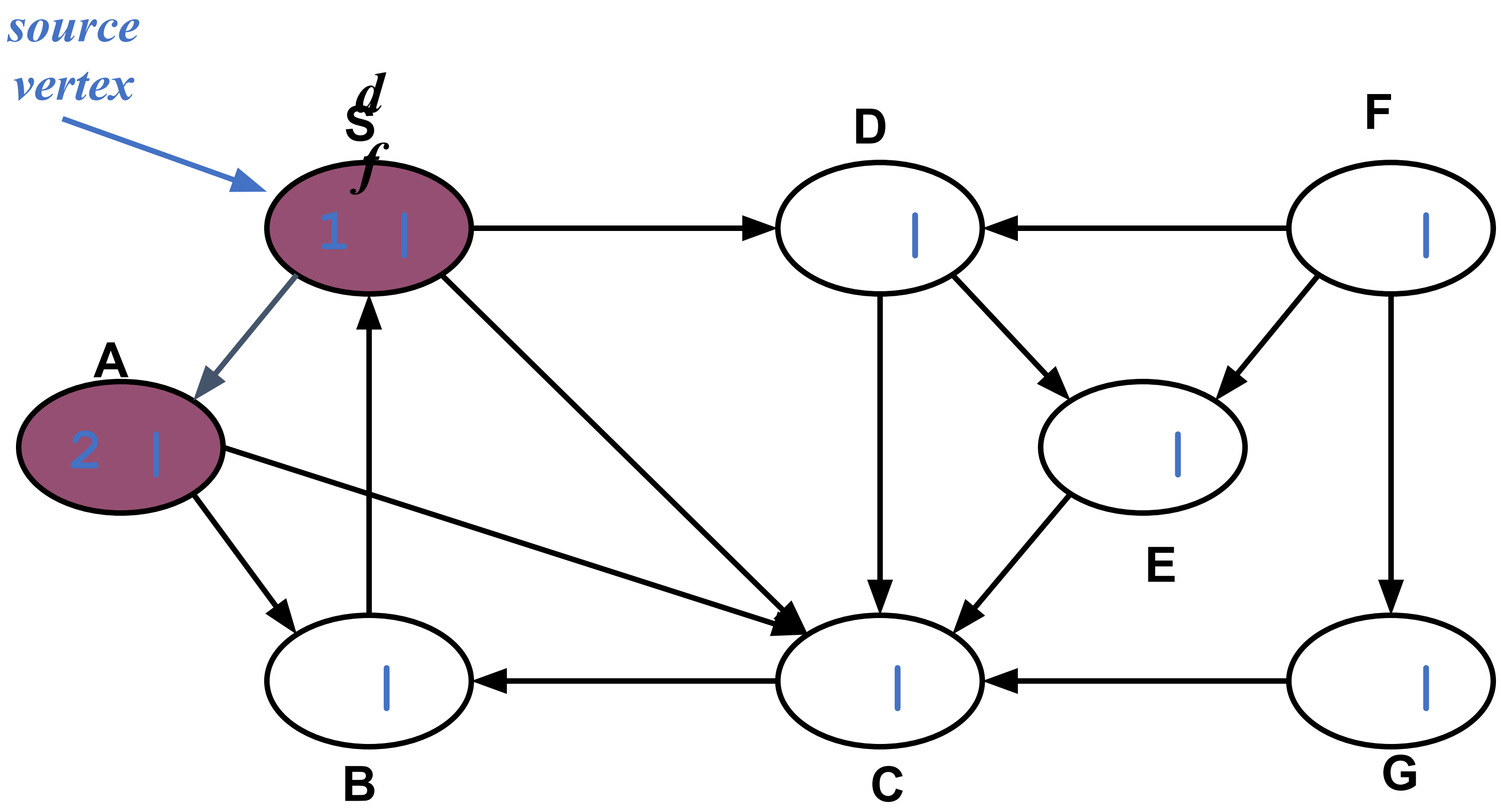
DFS Example



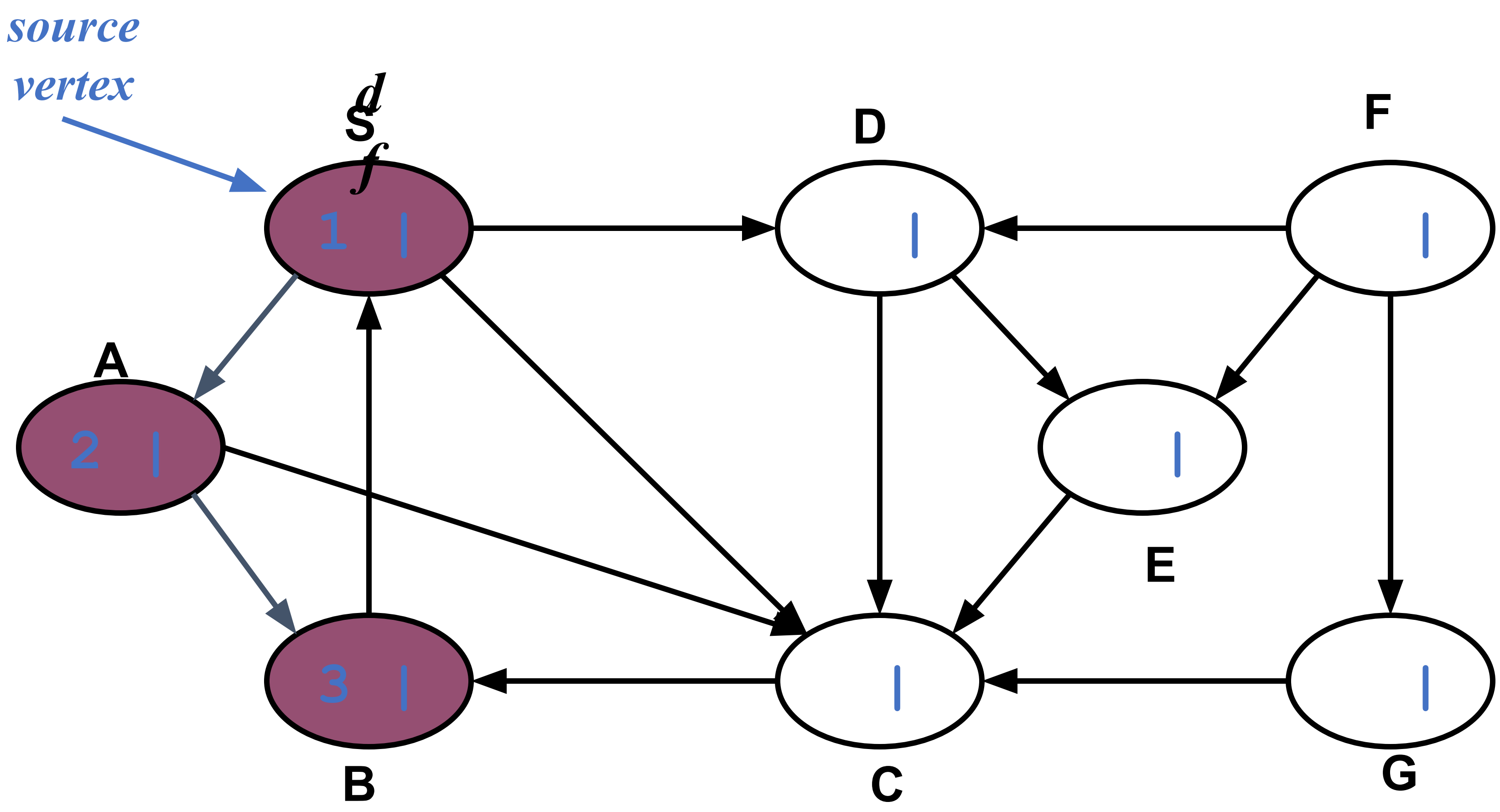
DFS Example



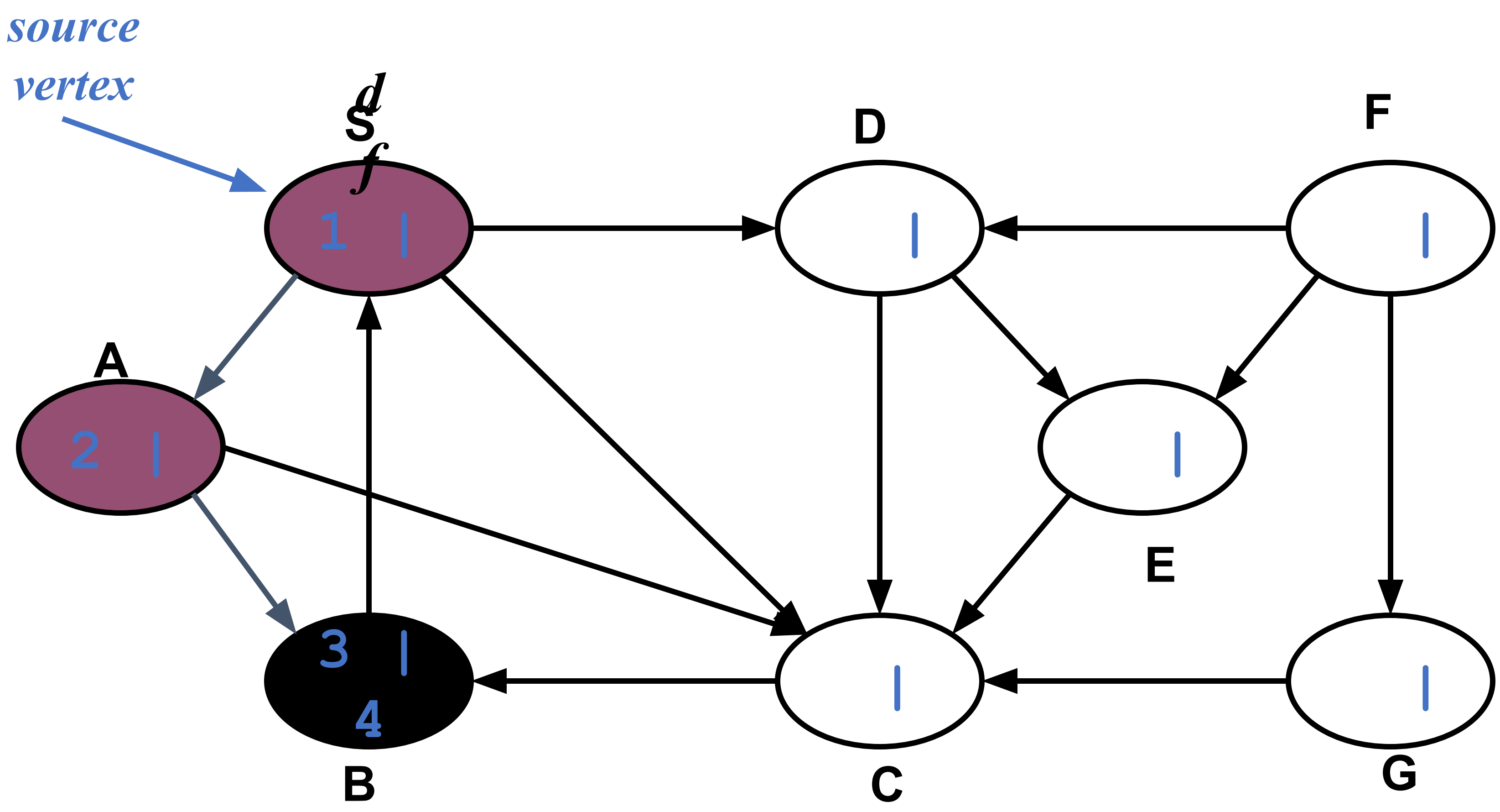
DFS Example



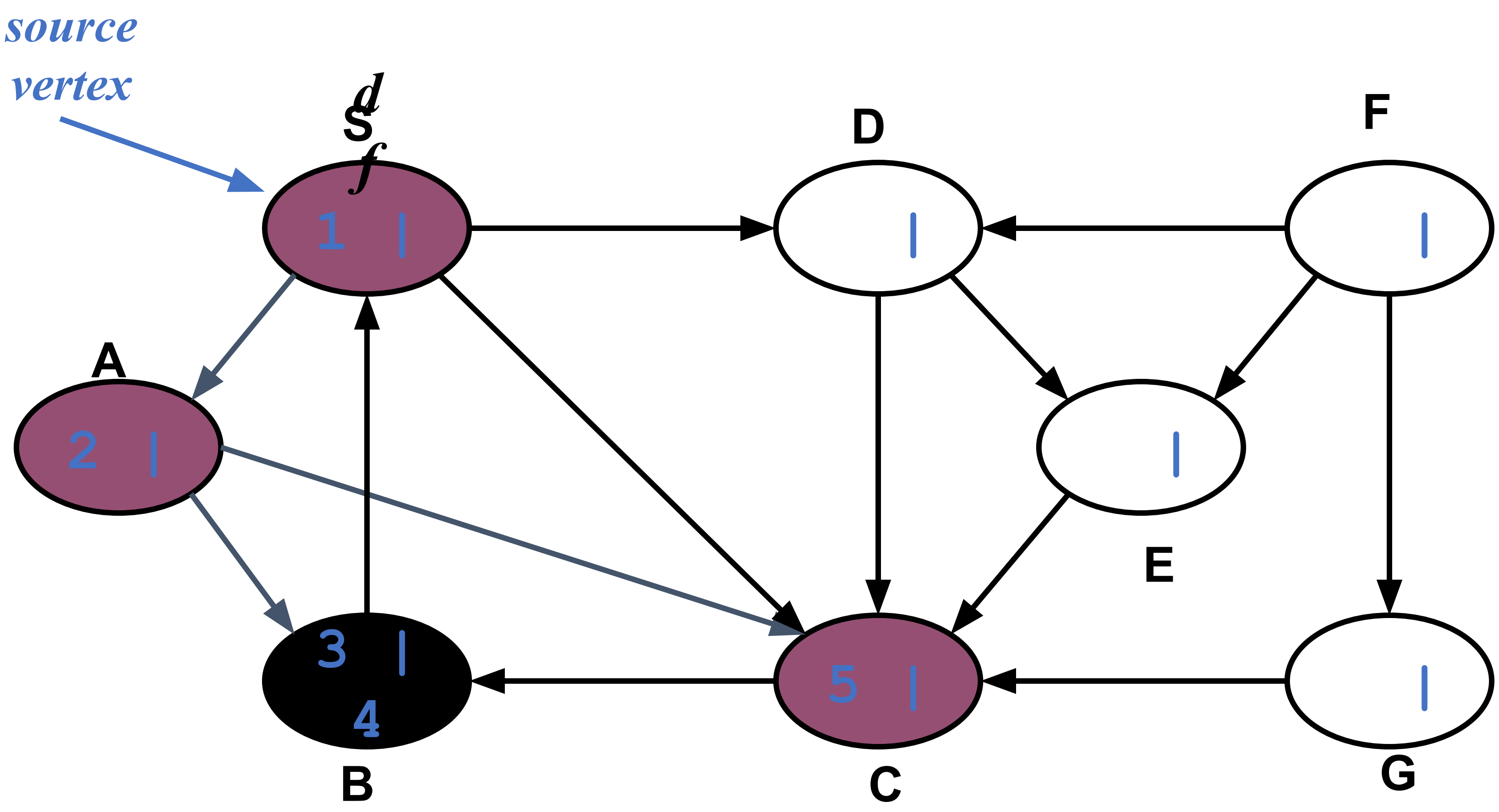
DFS Example



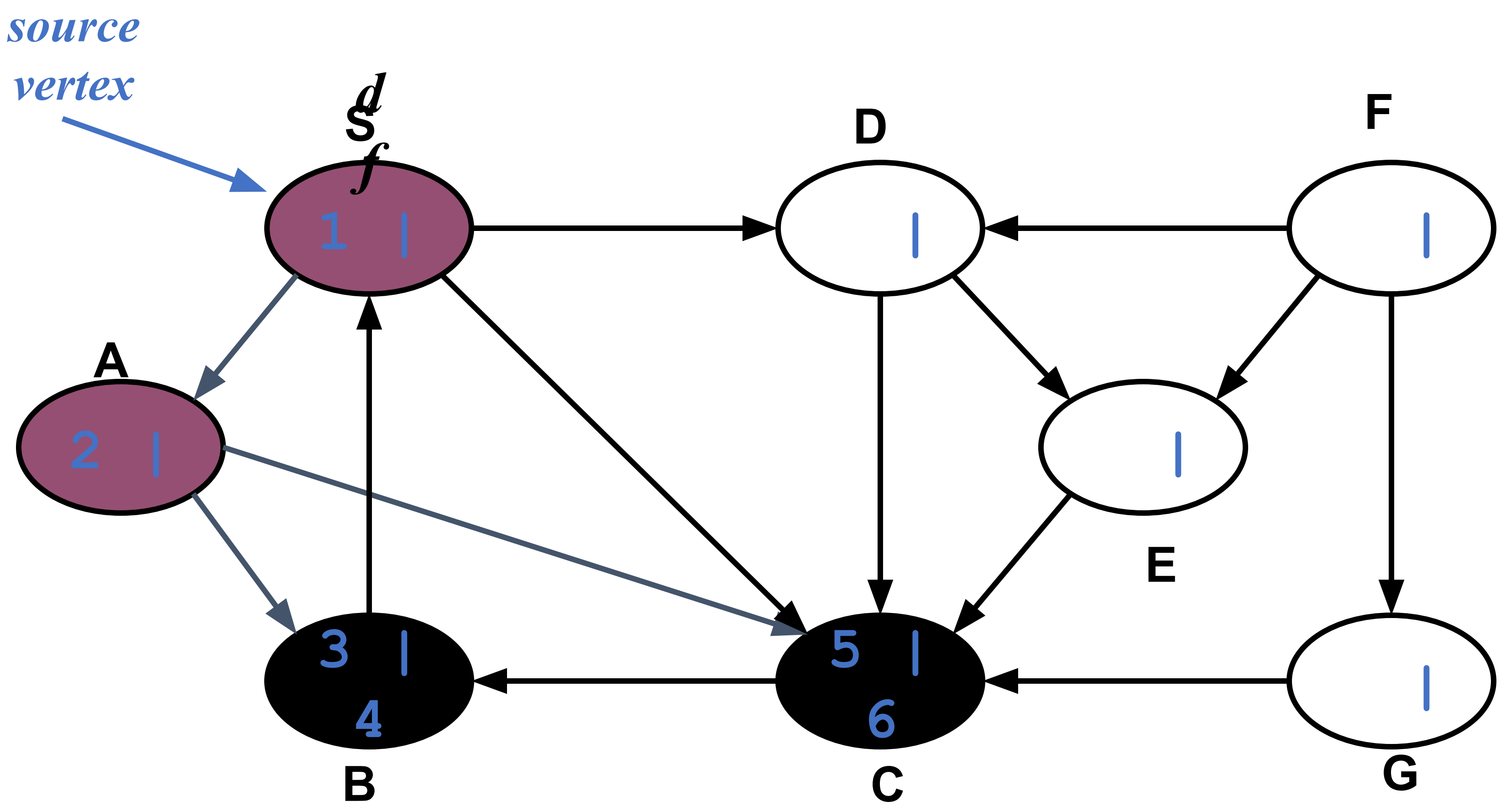
DFS Example



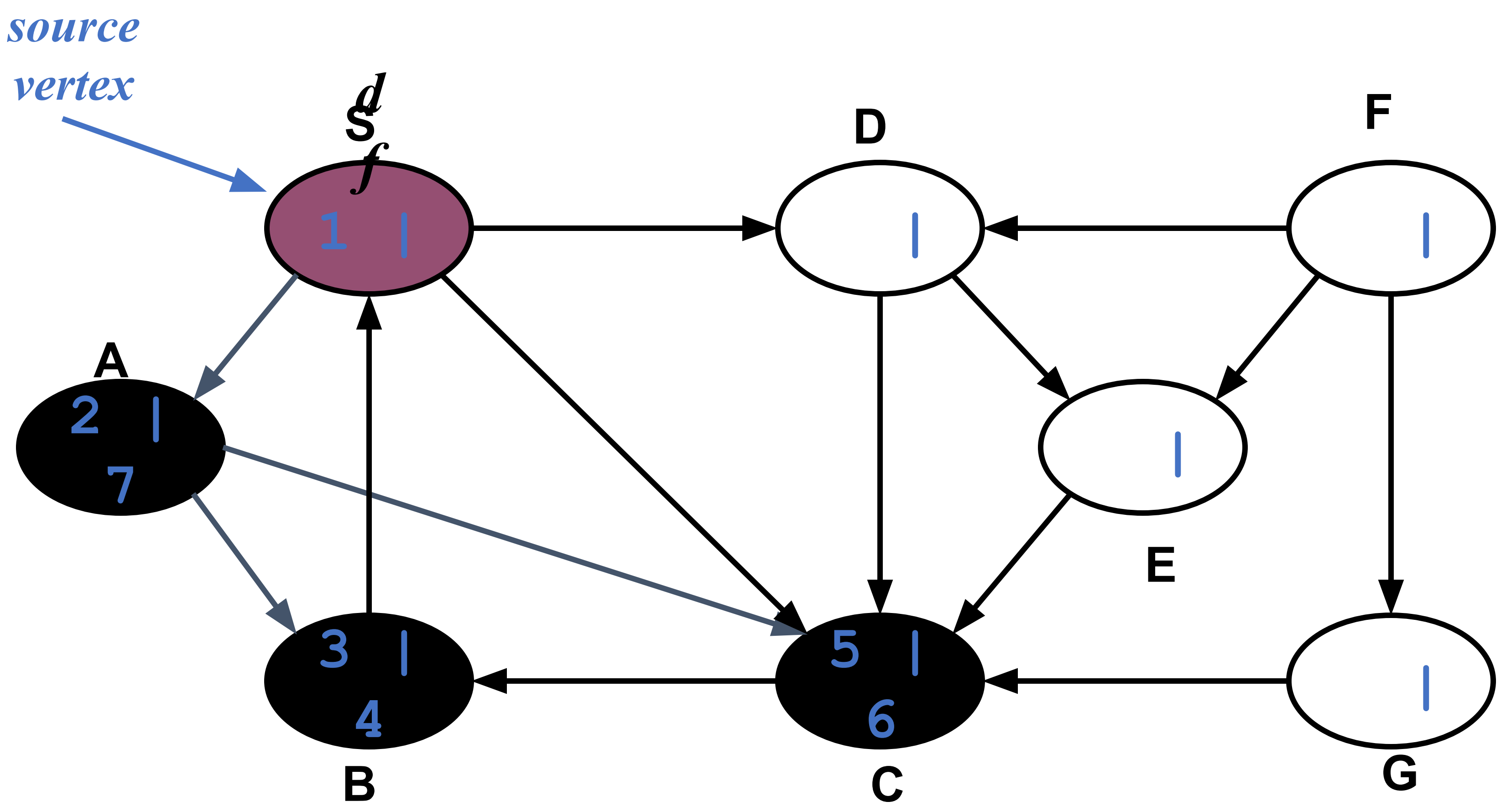
DFS Example



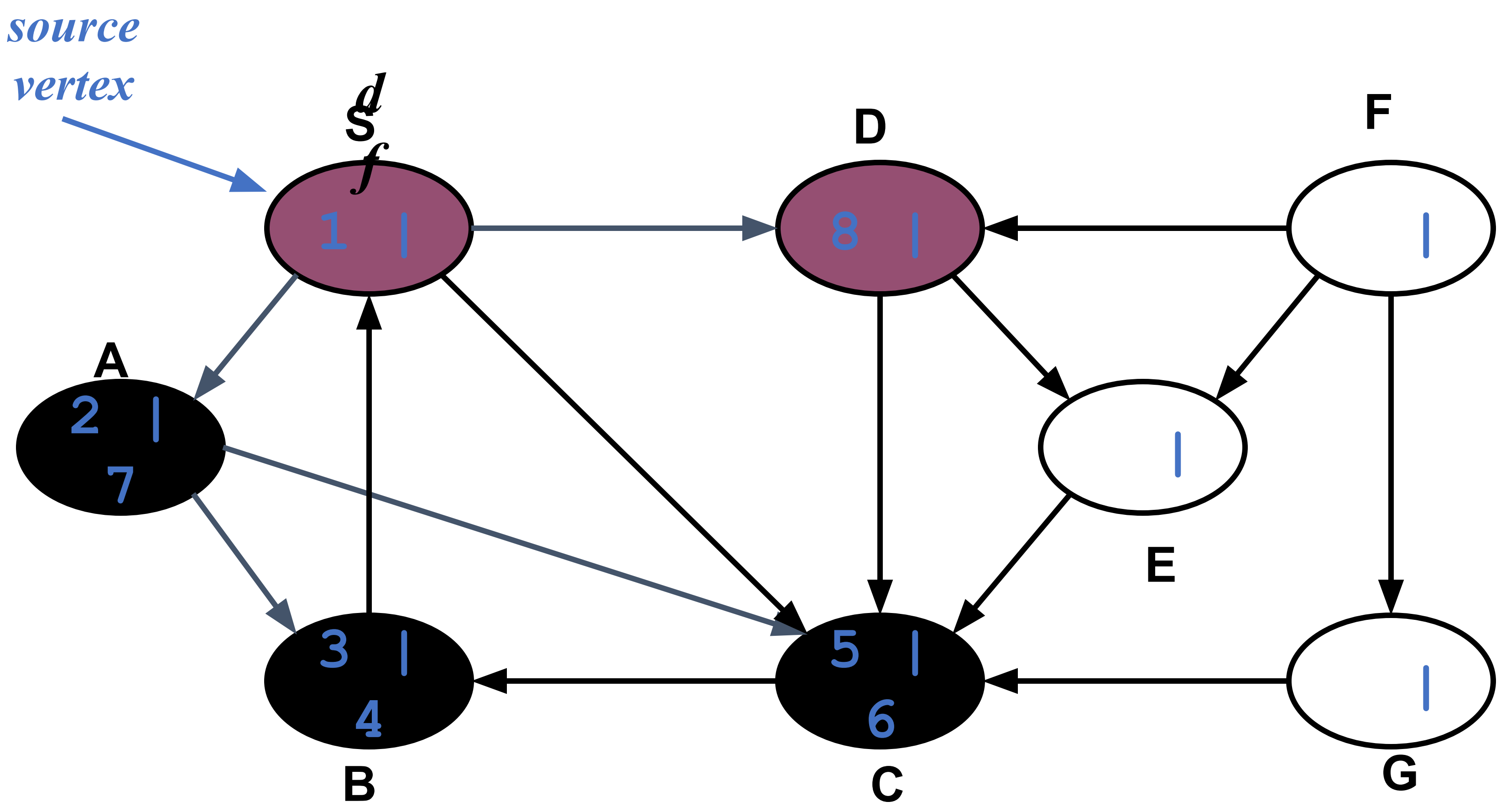
DFS Example



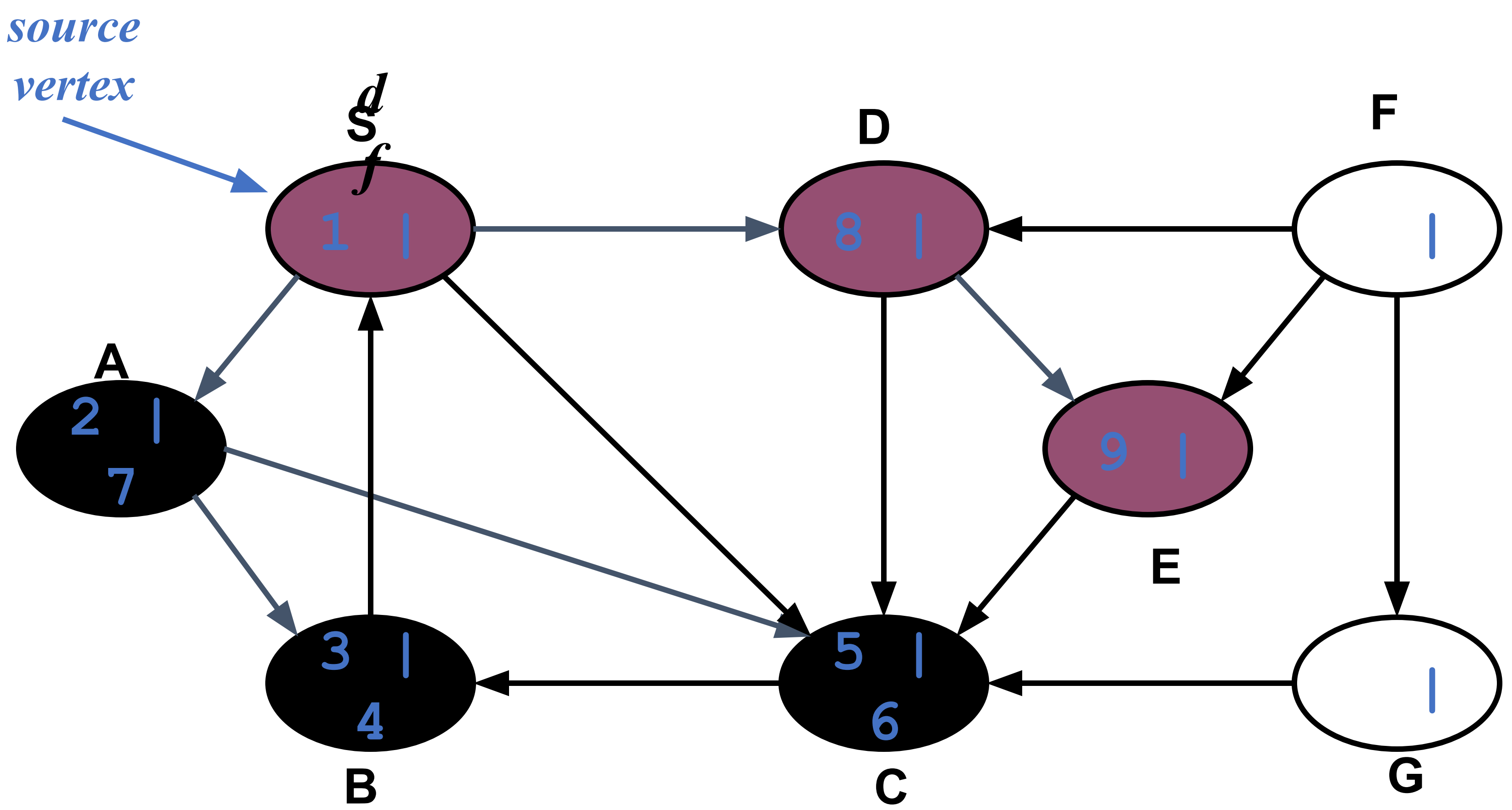
DFS Example



DFS Example

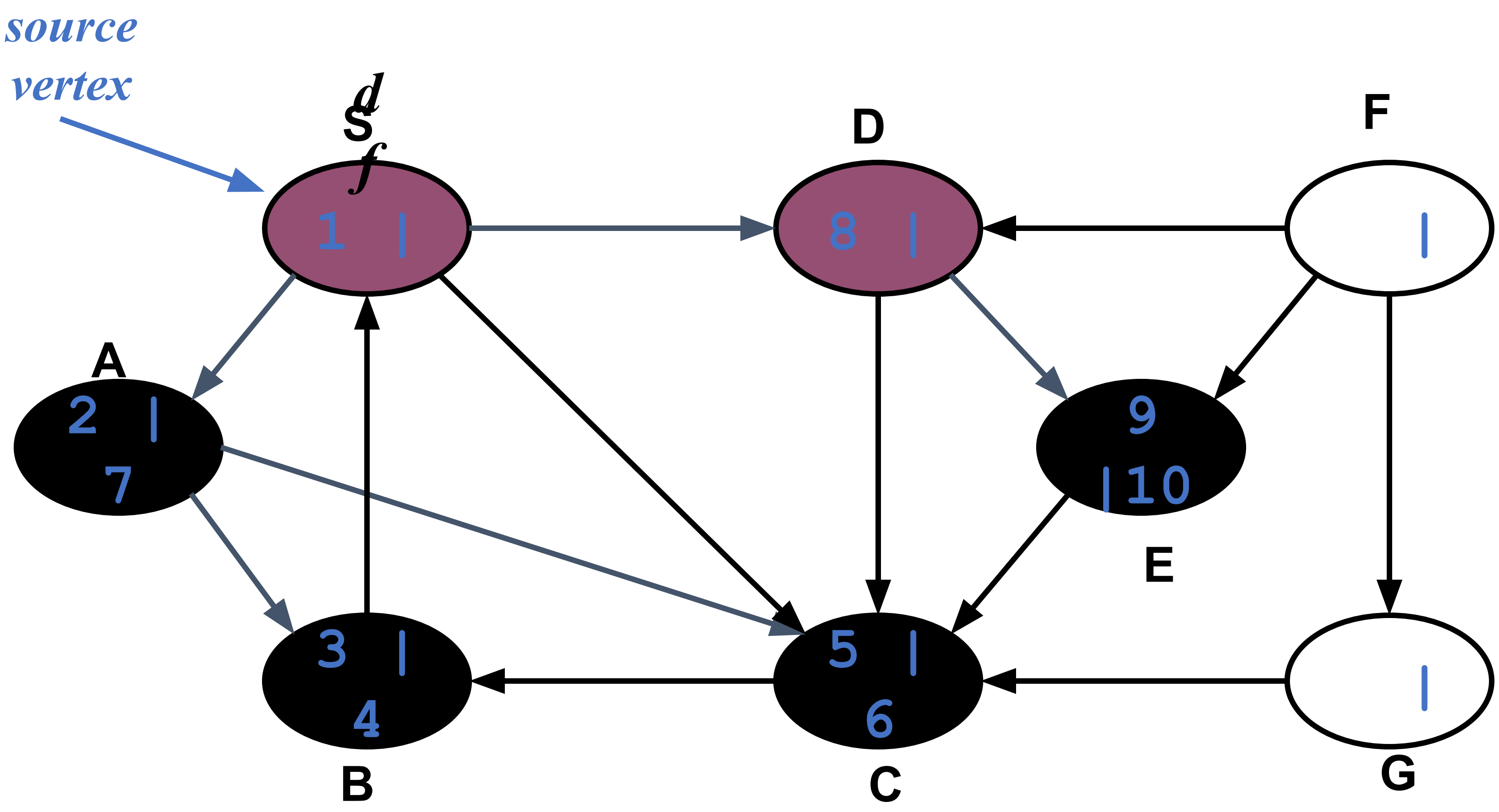


DFS Example

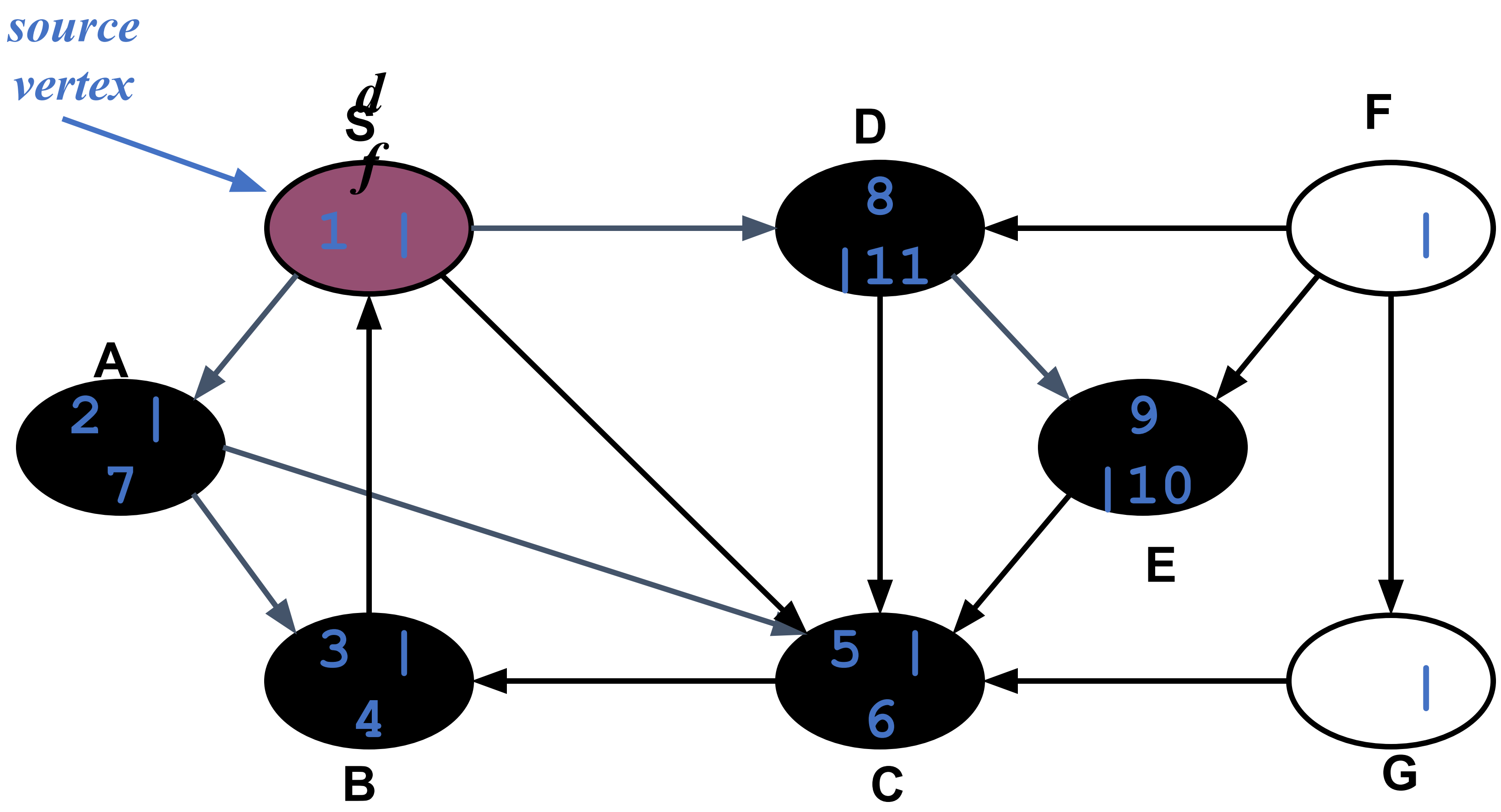


What is the structure of the grey vertices?
What do they represent?

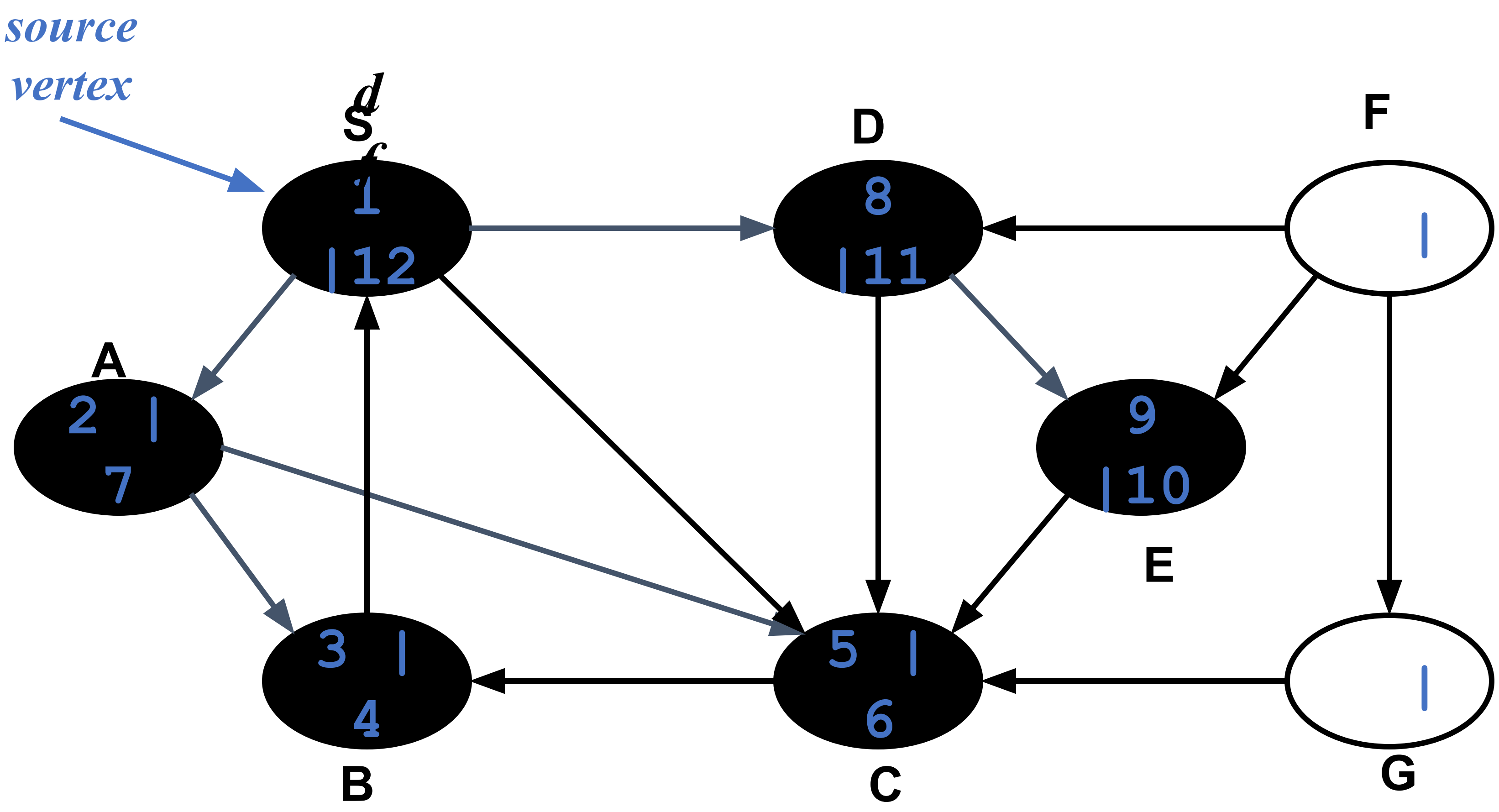
DFS Example



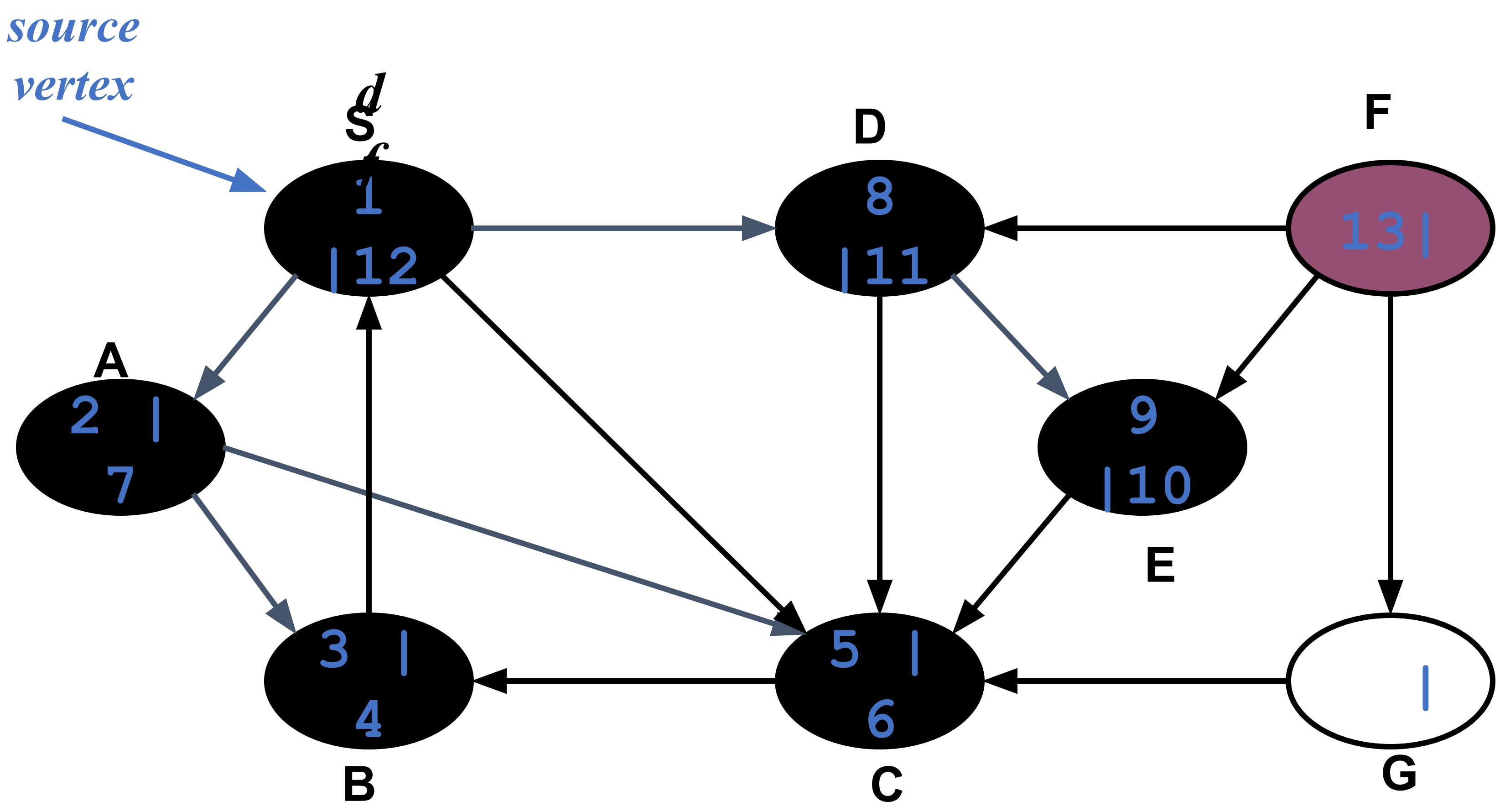
DFS Example



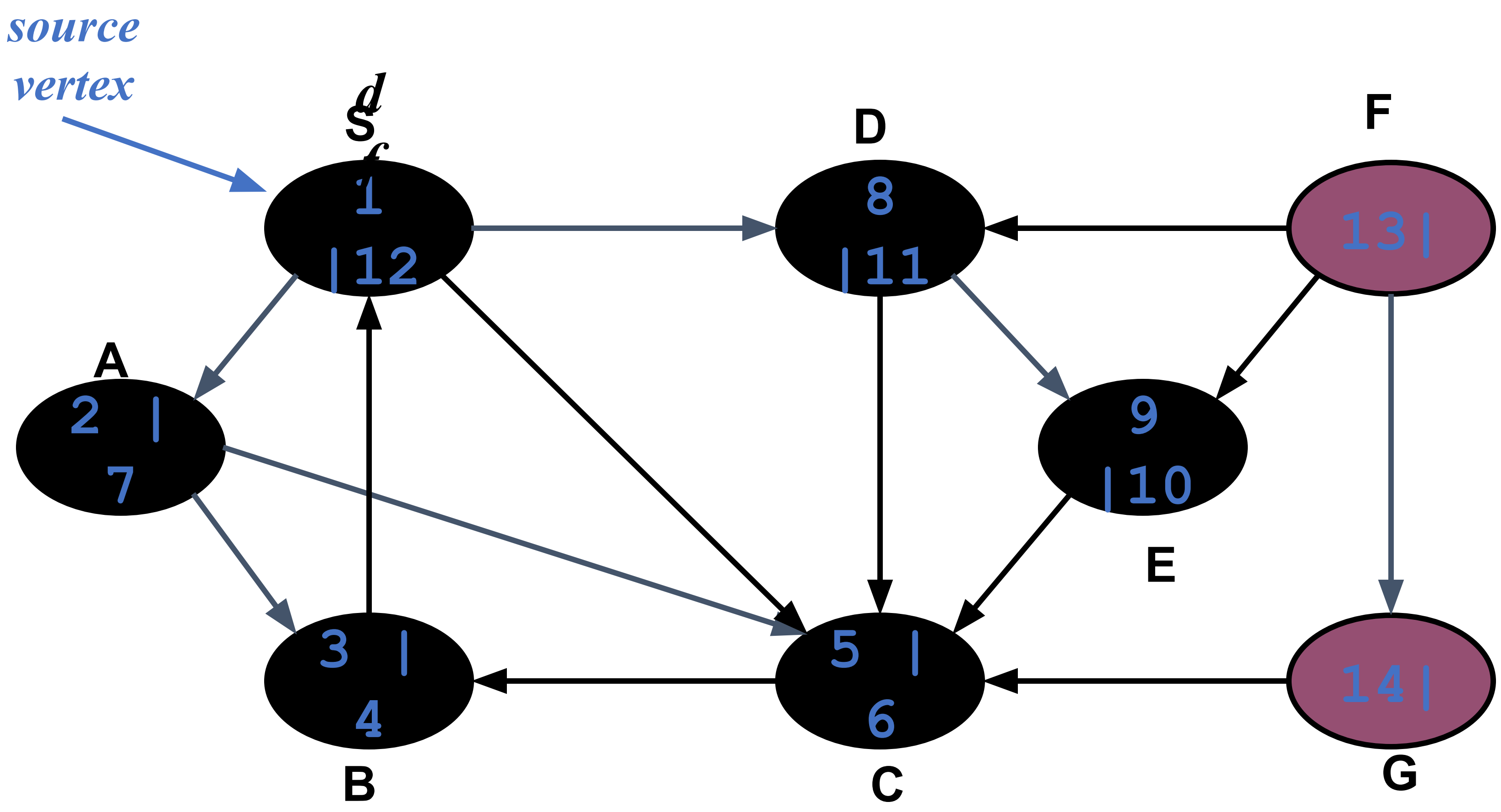
DFS Example



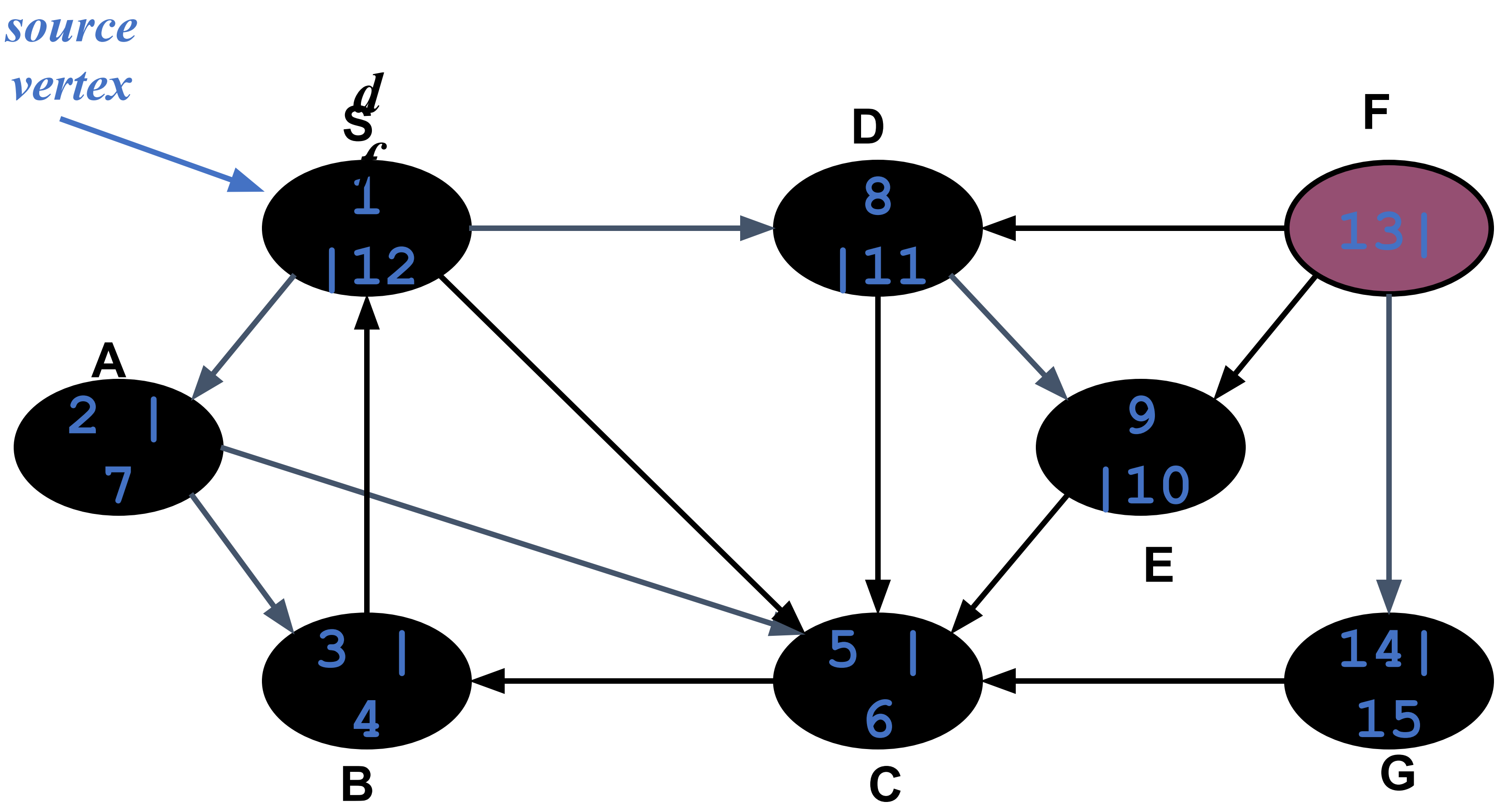
DFS Example



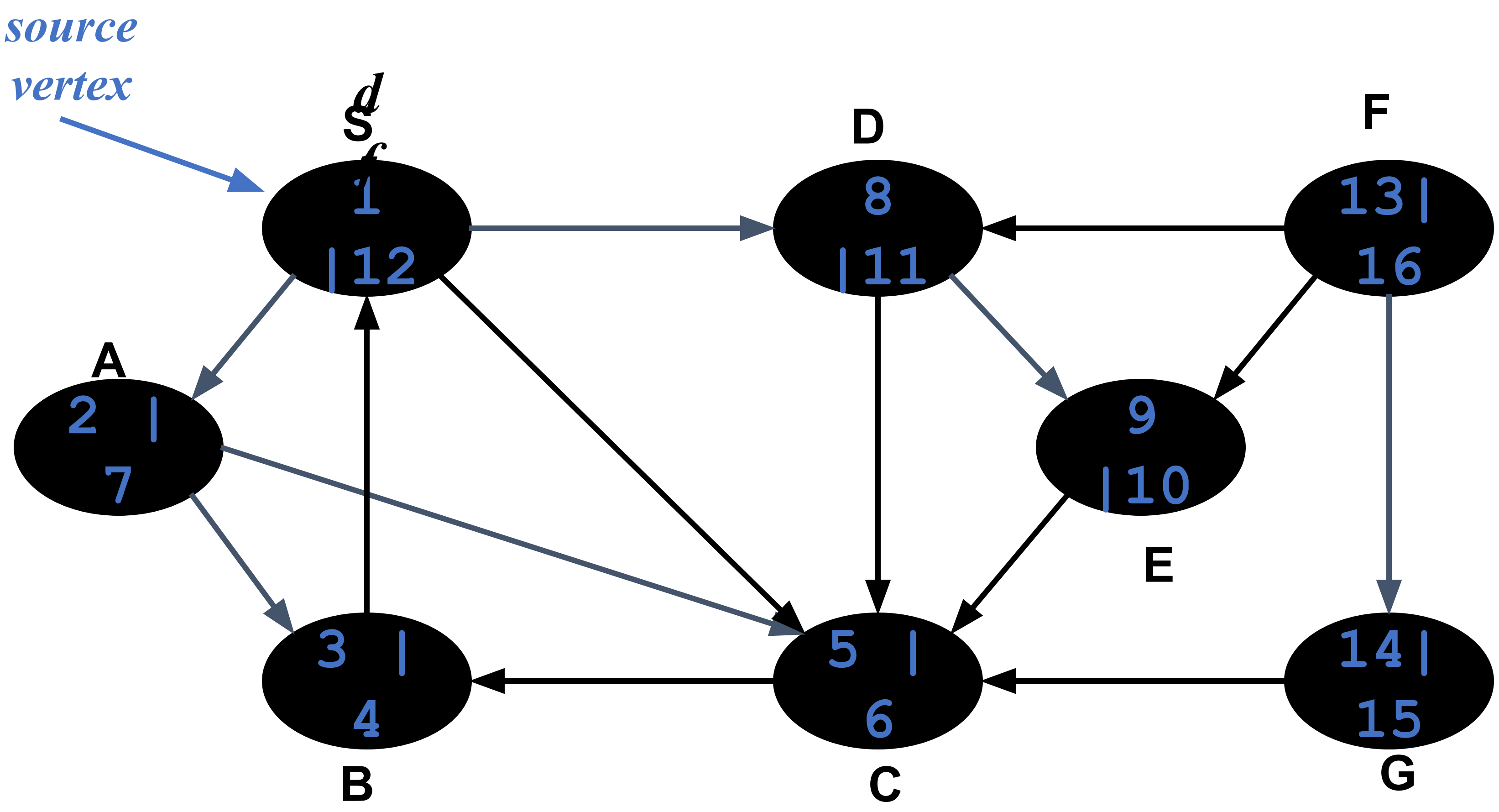
DFS Example



DFS Example



DFS Example



Depth-First Search: The Code

```
Data: color[V], time,
  prev[V],d[V], f[V]
DFS(G) // where prog starts
{
   for each vertex u ∈ V
   {
      color[u] = WHITE;
   prev[u]=NIL;
   f[u]=inf; d[u]=inf;
   }
   time = 0;
   for each vertex u ∈ V
     if (color[u] == WHITE)
         DFS_Visit(u);
}
```

```
DFS_Visit(u)
{
   color[u] = GREY;
   time = time+1;
   d[u] = time;
   for each v ∈ Adj[u]
   {
      if (color[v] == WHITE)
      prev[v]=u;
         DFS_Visit(v);
   }
   color[u] = BLACK;
   time = time+1;
   f[u] = time;
}
```

What will be the running time?

Depth-First Search: The Code

```
Data: color[V], time,
  prev[V],d[V], f[V]
DFS(G) // where prog starts
{
   for each vertex u ∈ V
   {
      color[u] = WHITE;
   prev[u]=NIL;
   f[u]=inf; d[u]=inf;
   }
   time = 0;
   for each vertex u ∈ V
     if (color[u] == WHITE)
        DFS_Visit(u);
}
```

O(V)

O(V)

```
DFS_Visit(u)
{
   color[u] = GREY;
   time = time+1;
   d[u] = time;
   for each v ∈ Adj[u]
   {
      if (color[v] == WHITE)
      prev[v]=u;
         DFS_Visit(v);
   }
   color[u] = BLACK;
   time = time+1;
   f[u] = time;
}
```

O(V)

Running time: $O(V^2)$ because call DFS_Visit on each vertex, and the loop over Adj[] can run as many as |V| times

Depth-First Search: The Code

```
Data: color[V], time,
  prev[V],d[V], f[V]
DFS(G) // where prog starts
{
    for each vertex u ∈ V
    {
        color[u] = WHITE;
    prev[u]=NIL;
    f[u]=inf; d[u]=inf;
    }
    time = 0;
    for each vertex u ∈ V
      if (color[u] == WHITE)
          DFS_Visit(u);
}
```

```
DFS_Visit(u)
{
    color[u] = GREY;
    time = time+1;
    d[u] = time;
    for each v ∈ Adj[u]
    {
       if (color[v] == WHITE)
       prev[v]=u;
          DFS_Visit(v);
    }
    color[u] = BLACK;
    time = time+1;
    f[u] = time;
}
```

BUT, there is actually a tighter bound.

How many times will DFS_Visit() actually be called?

Depth-First Search: The Code

```
Data: color[V], time,
  prev[V],d[V], f[V]
DFS(G) // where prog starts
{
   for each vertex u ∈ V
   {
      color[u] = WHITE;
   prev[u]=NIL;
   f[u]=inf; d[u]=inf;
   }
   time = 0;
   for each vertex u ∈ V
     if (color[u] == WHITE)
         DFS_Visit(u);
}
```

```
DFS_Visit(u)
{
   color[u] = GREY;
   time = time+1;
   d[u] = time;
   for each v ∈ Adj[u]
   {
      if (color[v] == WHITE)
      prev[v]=u;
         DFS_Visit(v);
   }
   color[u] = BLACK;
   time = time+1;
   f[u] = time;
}
```

So, running time of DFS = O(V+E)

DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex
    - The tree edges form a spanning forest
    - *Can tree edges form cycles? Why or why not?*
      - *No*

Textbooks & Web References

- Text Book (Chapter 22)
- www.geeksforgeeks.org

Thank you

&

Any question?